LNCS 3267

Corrado Priami
Paola Quaglia (Eds.)

Global Computing

IST/FET International Workshop, GC 2004
Rovereto, Italy, March 2004
Revised Selected Papers

Springer

Lecture Notes in Computer Science 3267

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Corrado Priami   Paola Quaglia (Eds.)

Global Computing

IST/FET International Workshop, GC 2004
Rovereto, Italy, March 9-12, 2004
Revised Selected Papers

Springer

Volume Editors

Corrado Priami
Paola Quaglia
Dip. Informatica e Telecomunicazioni, Univ. Trento
Via Sommarive 14, 38050 Povo (TN), Italy
E-mail: {priami, quaglia}@dit.unitn.it

Library of Congress Control Number: Applied for

CR Subject Classification (1998): C.2.4, D.1.3, D.2, D.4.6, F.2.1-2, I.2.11, D.3, F.3

ISSN 0302-9743
ISBN 3-540-24101-9 Springer Berlin Heidelberg New York

Springer is a part of Springer Science+Business Media

springeronline.com

Printed in Germany

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11353478 06/3142 5 4 3 2 1 0

Preface

This volume collects revised versions of some of the papers presented at the Second IST/FET International Workshop on Global Computing held in Rovereto, Italy (9–12 March, 2004).

The workshop involved all the thirteen projects funded under the IST/FET proactive initiative on GLOBAL COMPUTING: AGILE; CRESCCO; DART; DBGLOBE; DEGAS; FLAGS; MIKADO; MRG; MYTHS; PEPITO; PROFUNDIS; SECURE; SOCS.

The first aim of the GLOBAL COMPUTING initiative is the development of paradigms for building flexible, dependable, secure, robust and efficient systems. Primary research concerns are the co-ordination, interaction, security, reliability, robustness, and risk control of the entities in the global system. The ultimate goal of the research action is to provide a solid scientific foundation for the design of such systems, and to lay the groundwork for achieving effective principles for building and analysing them.

The workshop covered topics related to programming environments, dynamic reconfiguration, resource guarantees, peer-to-peer networks, analysis of systems and resources, resource sharing, and security, as well as foundational calculi for mobility. The present collection offers a rich sample of research results on the above subjects.

We acknowledge the Dipartimento di Informatica e Telecomunicazioni of the University of Trento for partially funding the workshop, and the Events and Meetings Office of the University of Trento for the valuable collaboration.

Trento
27 July 2004

Corrado Priami and Paola Quaglia

Table of Contents

Symbolic Equivalences for Open Systems*

Paolo Baldan[1], Andrea Bracciali[2], and Roberto Bruni[2]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italia
[2] Dipartimento di Informatica, Università di Pisa, Italia
baldan@dsi.unive.it, {braccia, bruni}@di.unipi.it

Abstract. Behavioural equivalences on open systems are usually defined by comparing system behaviour in all environments. Due to this "universal" quantification over the possible hosting environments, such equivalences are often difficult to check in a direct way. Here, working in the setting of process calculi, we introduce a hierarchy of behavioural equivalences for open systems, building on a previously defined symbolic approach. The hierarchy comprises both branching, bisimulation-based, and non-branching, trace-based, equivalences. Symbolic equivalences are amenable to effective analysis techniques (e.g., the symbolic transition system is finitely branching under mild assumptions), which result to be sound, but often not complete due to redundant information. Two kinds of redundancy, syntactic and semantic, are discussed and and one class of symbolic equivalences is identified that deals satisfactorily with syntactic redundant transitions, which are a primary source of incompleteness.

1 Introduction

The widespread diffusion of web applications and mobile devices has shifted the attention to *open systems*, i.e., systems where mobile software components can be dynamically connected to interact with each other. As a consequence, language-independent frameworks to reason about open systems and software architectures for coordination have gained interest. In the literature, process calculi (PC) have been devised as a useful paradigm for the specification and analysis of open systems. Situated between real programming languages and mere mathematical abstractions, they facilitate rigorous system analysis, offering the basis for prototypical implementation and for verification tools. Indeed, many running implementations exist of languages based on calculi originally proposed to experiment basic interaction primitives [29, 33, 16, 11].

The operational and abstract semantics of PC, as well as algorithms for verification, are often naturally defined for *components*, i.e. closed terms of the calculus, via a labelled transition system (LTS). The extension to *coordinators*, i.e. contexts with holes representing the openness of the system, can require

* Research supported by the Projects IST-2001-32747 Agile, IST-2001-32617 Myths and IST-2001-32530 Socs.

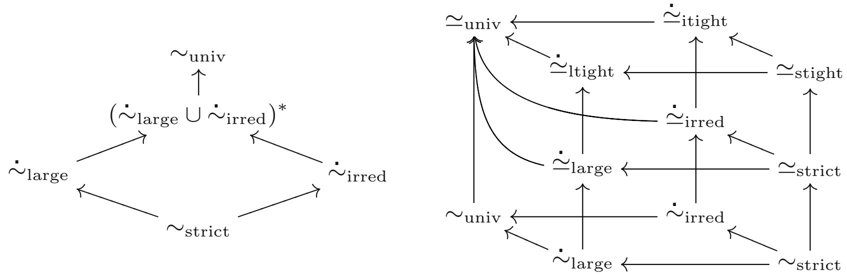
C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 1–17, 2004.

non-trivial enhancements. Here, in the style, e.g., of the Security Process Algebra (see [19] and references therein), we use process variables as place-holders for unspecified components which may join the system, i.e., coordinators are viewed as terms with process variables. A way to lift a semantic equivalence $\approx$ from components to coordinators is to define $C[X] \approx_{\text{univ}} D[X]$ when $C[p] \approx D[p]$ for all components p. This universal quantification can be recasted in a coalgebraic framework by enriching the original transition system over components: all coordinators $C[X]$ are taken as states and a transition $C[X] \rightarrow_p C[p]$ is added for any component p. However the extended transition system is likely to be intractable, being infinitely branching even for trivial calculi.

In [5], we introduced *symbolic transition systems* (STSs) to ease the analysis of coordinators' properties. An STS is a transition system where states are coordinators and transition labels are logic formulae expressing structural and behavioural requirements on the unknown components which would allow the transition to occur. Symbolic transition systems account for the operational semantics of coordinators, and, based on this, two abstract semantics are defined: *strict bisimilarity* $\sim_{\text{strict}}$, a straight extension of the standard bisimilarity on labelled transition systems, and *large bisimilarity* $\dot{\sim}_{\text{large}}$, introduced as a mean to solve, at least in part, the problem of redundant symbolic transitions (see below) which may cause $\sim_{\text{strict}}$ to distinguish "too much". For suitable STSs (i.e., sound and complete w.r.t. the original LTS) both "symbolic" bisimilarities approximate $\sim_{\text{univ}}$, the standard extension of bisimilarity $\sim$ to coordinators defined by universal quantification, as illustrated above. Moreover, sound and complete STSs can be automatically derived from SOS specifications whenever the SOS rules satisfy rather general syntactic formats.

The first part of this paper consolidates and extends the theory of symbolic bisimilarities. More specifically, we investigate some basic properties of $\sim_{\text{strict}}$ and $\dot{\sim}_{\text{large}}$, showing, e.g., that the latter approximates $\sim_{\text{univ}}$ strictly better than $\sim_{\text{strict}}$, although in general it is non-transitive (incidentally, the dot on top of $\sim$ in the symbol for large bisimilarity is a reminder of this fact). We discuss how the defined equivalences are influenced by *redundant* symbolic specifications, identifying two kinds of redundancy, called *syntactic* and *semantic*. While $\sim_{\text{strict}}$ cannot overcome redundancy at all, $\dot{\sim}_{\text{large}}$ can deal with significant forms of both kinds of redundancy, but it does not fully solve any of the two. This motivates the introduction of a novel bisimilarity $\dot{\sim}_{\text{irred}}$, called *irredundant*, which solves the problem of syntactic redundancy. In general $\dot{\sim}_{\text{large}}$ and $\dot{\sim}_{\text{irred}}$ are not comparable and a fourth, better approximation of $\sim_{\text{univ}}$ can be obtained by combining $\dot{\sim}_{\text{large}}$ and $\dot{\sim}_{\text{irred}}$, originating the "diamond" of bisimilarities in Fig. 1(a).

The second part of the paper fully generalises the STS approach to the setting of *trace semantics*. Albeit trace semantics are usually easier to deal with, in the case of coordinators the problem of universal closure w.r.t. all components still persists and thus also trace equivalences benefit from a symbolic approach. To every kind of symbolic bisimilarity described above, there corresponds a notion of symbolic trace semantics. Each trace semantics is refined by the corresponding bisimilarity, as expected, and all are correct approximations of the universal trace


$\sim_{\text{univ}}$
$(\dot\sim_{\text{large}} \cup \dot\sim_{\text{irred}})^*$
$\dot\sim_{\text{large}}$
$\dot\sim_{\text{irred}}$
$\sim_{\text{strict}}$
$\simeq_{\text{univ}}$
$\dot\simeq_{\text{itight}}$
$\dot\simeq_{\text{ltight}}$
$\simeq_{\text{stight}}$
$\dot\simeq_{\text{irred}}$
$\dot\simeq_{\text{large}}$
$\simeq_{\text{strict}}$
$\sim_{\text{univ}}$
$\dot\sim_{\text{irred}}$
$\dot\sim_{\text{large}}$
$\sim_{\text{strict}}$

(a) Diamond of bisimilarities.

(b) Tower of symbolic semantics.

Fig. 1.

equivalence $\simeq_{\text{univ}}$. Finally, we introduce a compact form of trace, called *tight*, which is exploited to improve the precision of all the approximations of $\simeq_{\text{univ}}$. Though symbolic trace equivalences are the natural counterparts of symbolic bisimilarities, the notion of *tight trace* is original and fully exploits the use of formulae as transition labels. The full hierarchy of equivalences is in Fig. 1(b).

Synopsis. § 2 recalls the principles of STSs from [5]. All material in § 3–5 is original to this contribution. Relations between $\sim_{\text{strict}}$ and $\dot\sim_{\text{large}}$ are investigated in § 3, while § 4 discusses syntactic and semantic redundancy. § 5 provides a symbolic approach to trace semantics. Technical results come together with examples, based on calculi designed ad-hoc to clarify the features of interest. Some concluding remarks and an account of related work are in § 6.

2 Approximating the Universal Bisimilarity

We restrict here to (non-empty) process calculi based on unsorted signatures. Given a process signature Σ and a denumerable set of variables $\mathcal{X}$ (disjoint from Σ), $\mathbb{T}_\Sigma(\mathcal{X})$ denotes the term-algebra over Σ with variables in $\mathcal{X}$. For $P \in \mathbb{T}_\Sigma(\mathcal{X})$, $var(P)$ denotes the set of variables in P. If $var(P) = \emptyset$ then P is *closed.* Closed terms form the set $\mathcal{P}$ of *components* p (possibly taken modulo a structural congruence $\equiv$), while terms in $\mathbb{T}_\Sigma(\mathcal{X})$ form the set $\mathcal{C}$ of *coordinators* C. With $C[X_1, \ldots, X_n]$ we mean that C is a coordinator such that $var(C) \subseteq \{X_1, \ldots, X_n\}$. To simplify the notation hereafter we shall use single-holed coordinators, i.e., coordinators with at most one variable, but all definitions and results straightforwardly extend to many-holed coordinators.

The operational semantics of process calculi is given in terms of *labelled transition systems* (LTSs). A transition from p to q with observable $a \in \Lambda$ (the label alphabet) is indicated as $p {\rightarrow_a} q$. Transitions are often specified by a collection of inductive rules, following the SOS paradigm [31]. Throughout the paper PC denotes a fixed process calculus over a signature Σ, with an LTS $\mathcal{L}$ specified by a set of SOS rules.

A *bisimulation* is a symmetric relation $\approx$ over components such that if $p \approx q$, then for any transition $p \rightarrow_a p'$ there exist a component q' and a transition $q \rightarrow_a q'$ with $p' \approx q'$. *Bisimilarity* $\sim$ is the largest bisimulation. The *universal bisimilarity* $\sim_{\mathrm{univ}}$ is the lifting of $\sim$ to coordinators obtained by closing under all substitutions, i.e. $C[X] \sim_{\mathrm{univ}} D[X] \overset{\mathrm{def}}{\iff} \forall p \in \mathcal{P},\ C[p] \sim D[p]$. Since components are closed, $p \sim_{\mathrm{univ}} q$ iff $p \sim q$.

Symbolic Bisimulation. The equivalence $\sim_{\mathrm{univ}}$ can be quite intractable. To address this problem we exploit a symbolic approach based on:

1. abstracting from components not playing an active role in the transition;
2. specifying the active components as little as possible;
3. making assumptions on the structure and behaviour of active components.

The idea is to derive from the LTS a *symbolic transition system* (STS), where states are coordinators and labels are formulae expressing behavioural and structural conditions required to unknown components for enabling the transition.

The logic L_{PC} that we consider has *modal* and *spatial* operators in the style of [8, 12]. It is worth observing that the word "spatial" has been used in the literature to refer to the logical or physical distribution of system components, e.g., prefix in CCS is generally not taken as a spatial operator. For the aim of this paper, this word refers to the structure of a term and any operator of the signature can be considered spatial. The syntax of L_{PC}-formulae φ and the associated notion of satisfaction are given below, where $X \in \mathcal{X}$ denotes a process variable, $f \in \Sigma$ is an operator in the process signature and $a \in \Lambda$ an action label.

$$\varphi ::= X \mid f(\varphi, \ldots, \varphi) \mid \diamond a.\, \varphi$$

$$\begin{array}{lll} p \models X & & \\ p \models f(\varphi_1, \ldots, \varphi_n) & \text{iff} & \exists p_1, \ldots, p_n.\ p \equiv f(p_1, \ldots, p_n)\ \wedge\ \forall i.\ p_i \models \varphi_i \\ p \models \diamond a.\, \varphi & \text{iff} & \exists p'.\ p \rightarrow_a p'\ \wedge\ p' \models \varphi \end{array}$$

We denote by $var(\varphi)$ the set of variables in a formula φ. We consider *linear* formulae only (i.e. formulae where no variable occurs twice). A formula in L_{PC} is called *spatial* if it only contains variables and spatial operators $f \in \Sigma$ (abusing the notation, spatial expressions can be read both as formulae and as coordinators). Each component p can be regarded as a spatial formula with no variables, and $p \models q$ iff $p \equiv q$.

For instance, the action prefix operator yields the spatial formula $a.X$, which is satisfied by components of the shape $p \equiv a.q$. Although for specific calculi the formulae $a.X$ and $\diamond a.X$ are satisfied exactly by the same set of components (e.g. the formulae $r.X$ and $\diamond r.X$ in Example 1), we remark that their meaning is quite different: the former imposes a spatial constraint, the latter imposes a behavioural constraint, satisfied by components which can perform the action a (e.g., the process $(b.0 \mid a.0)\backslash b$ in a CCS-like calculus).

Definition 1 (sts). *A* symbolic transition system *(*STS*) $\mathcal{S}$ for PC is a set of transitions $C[X] \overset{\{\varphi\}}{\dashrightarrow}_a D[Y]$ where $C[X]$ and $D[Y]$ are coordinators in PC, $a \in \Lambda$ and φ is a formula in L_{PC} with $var(\varphi) \supseteq var(D)$.*

The correspondence between the variable X in the source and its residual Y in the target is expressed by the occurrence of Y in φ. For example, a symbolic transition in a CCS-like calculus could be $X\backslash b \;\text{-}\{\Diamond a.Y\}\!\!\rightarrow_a Y\backslash b$ for $a \neq b$. The modal formula $\Diamond a.Y$ is satisfied by any process p which can perform an action a becoming a generic process, say q. Hence the symbolic transition represents the infinitely many concrete transitions $p\backslash b \rightarrow_a q\backslash b$ which are obtained by replacing X and Y by such p and q, respectively.

To provide an adequate representation of the original transition system, an STS is required to satisfy suitable correspondence properties. Informally, $C[X] \;\text{-}\{\varphi\}\!\!\rightarrow_a D[Y]$ means that the coordinator C, instantiated with any component p satisfying φ, i.e., $p \models \varphi[q/Y]$, must be able to perform the action a becoming an instance of D, namely $D[q]$. Also, any concrete transition on components should have symbolic counterparts.

Definition 2 (Sound/Complete sts). *An* STS $\mathcal{S}$ *for* PC *is:*

- sound*, if for any symbolic transition* $C[X] \;\text{-}\{\varphi\}\!\!\rightarrow_a D[Y]$ *in* $\mathcal{S}$ *and for any* p, q *with* $p \models \varphi[q/Y]$*, there exists a transition* $C[p] \rightarrow_a D[q]$ *in the* LTS *of* PC.
- complete*, if for any coordinator* $C[X]$*, for any* p *and for any transition* $C[p] \rightarrow_a r$ *in* PC *there are* q *and* $C[X] \;\text{-}\{\varphi\}\!\!\rightarrow_a D[Y]$ *in* $\mathcal{S}$ *with* $p \models \varphi[q/Y]$ *and* $r \equiv D[q]$.

Observe that a weaker notion of completeness, simply asking that for any $p \rightarrow_a q$ there exist $C[X] \;\text{-}\{\varphi\}\!\!\rightarrow_a D[Y]$ and p', q' such that $C[p'] \equiv p$, $D[q'] \equiv q$ and $p' \models \varphi[q'/Y]$ would be inappropriate since a complete STS would not represent the proper computational behaviour of coordinators. For instance, it is easy to see that the LTS of components (seen as a trivial STS) would be complete according to the weaker notion of completeness, although it does not include any transition for terms with variables.

The straightforward definition of bisimulation equivalence over an STS is given below.

Definition 3 ($\sim_{\text{strict}}$). *A symmetric relation* $\approx$ *on coordinators is a* strict symbolic bisimulation *if for all* $C[X]$, $D[X]$ *with* $C[X] \approx D[X]$ *and for any symbolic transition* $C[X] \;\text{-}\{\varphi\}\!\!\rightarrow_a C'[Y]$*, there exists* $D[X] \;\text{-}\{\varphi\}\!\!\rightarrow_a D'[Y]$ *such that* $C'[Y] \approx D'[Y]$*. The largest strict symbolic bisimulation* $\sim_{\text{strict}}$ *is an equivalence called* strict symbolic bisimilarity

Strict bisimilarity requires a transition to be simulated by a transition with exactly the same label. Syntactic equality has been preferred to logical equivalence since, in general, the latter could be hard to verify or, even worse, undecidable. Nevertheless, given a specific calculus, equivalences which are easy to check can be exploited in symbolic bisimilarity (e.g., to standardise the labels) and the theory easily carries over.

Strict symbolic bisimilarity distinguishes at least as much as universal bisimilarity, i.e. $\sim_{\text{strict}}$ implies $\sim_{\text{univ}}$ (Theorem 1 below, taken from [5]), but the converse does not hold in general. A better approximation of $\sim_{\text{univ}}$ is obtained by relaxing the requirement of exact (spatial) matching between formulae.

Definition 4 ($\dot{\sim}_{\text{large}}$). *A symmetric relation $\approx$ on coordinators is a* large symbolic bisimulation *if for all $C[X]$, $D[X]$ with $C[X] \approx D[X]$ and for any transition $C[X] \xrightarrow{\varphi}_a C'[Y]$ there exist a transition $D[X] \xrightarrow{\psi}_a D'[Z]$ and a spatial formula η such that $\varphi = \psi[\eta/Z]$ and $C'[Y] \approx D'[\eta]$. The greatest large bisimulation $\dot{\sim}_{\text{large}}$ is called* large symbolic bisimilarity*.*

As a trivial example, let $\Sigma = \{a, f(.), g(.)\}$ and take the STS with transitions $f(X) \xrightarrow{Y}_\tau Y$, $g(X) \xrightarrow{Y}_\tau Y$, and $g(X) \xrightarrow{a}_\tau a$. Obviously, $f(X) \not\sim_{\text{strict}} g(X)$, because $f(X)$ cannot match the last transition of $g(X)$, while $f(X) \dot{\sim}_{\text{large}} g(X)$ since the formula X is "more general" than the spatial formula a.

Theorem 1 ($\sim_{\text{strict}} \Rightarrow \dot{\sim}_{\text{large}} \Rightarrow \sim_{\text{univ}}$). *For any sound and complete* STS *and for all coordinators $C[X]$, $D[X]$ we have*

$$C[X] \sim_{\text{strict}} D[X] \quad \Rightarrow \quad C[X] \dot{\sim}_{\text{large}} D[X] \quad \Rightarrow \quad C[X] \sim_{\text{univ}} D[X].$$

Bisimulation by Unification. The framework introduced in [5] is completed by a constructive definition of a suitable STS associated to any PC whose operational proof rules are in *algebraic format* [20] (that generalises, e.g., the well-known De Simone format [17]). Starting from the algebraic SOS proof rules for PC, a Prolog program $\mathit{Prog}_{\mathrm{A}}(\mathsf{PC})$ can be derived which specifies a *sound* and *complete* STS over L_{PC}. The program defines a predicate `trs(X,A,Y)` whose intended meaning is "any component satisfying X can perform a transition labelled by A and become a component satisfying Y". Then, given a coordinator $C[X]$, if the query `?- trs(C[X], A, Z)` is successful, the corresponding computed answer substitution represents a symbolic transition for the coordinator. The code in $\mathit{Prog}_{\mathrm{A}}(\mathsf{PC})$ consists of the obvious translation of the SOS rules into Horn clauses, with an additional rule to handle behavioural formulae. Intuitively, the unification mechanism is used to compute the minimal requirements on the process variables of a coordinator which allow an SOS rule to be applied. We remark that if the set of SOS rules of PC is finite, then the program $\mathit{Prog}_{\mathrm{A}}(\mathsf{PC})$ has a finite number of clauses and the defined STS is finitely branching (even if the whole STS has instead infinitely many states and transitions, as obviously it must include all the original transitions over components).

3 Properties of Strict and Large Bisimilarities

In this section we study some basic properties of $\sim_{\text{strict}}$ and $\dot{\sim}_{\text{large}}$, and we show, by means of a few examples, that they capture different notions of simulation.

Comparing $\sim_{\text{strict}}$ and $\dot{\sim}_{\text{large}}$. The relation $\dot{\sim}_{\text{large}}$ is always coarser than $\sim_{\text{strict}}$. On the other hand, $\dot{\sim}_{\text{large}}$ is *not* guaranteed to be an equivalence relation, since it may fail to be transitive in some "pathological" situations (as the one below).

Example 1. Consider the simple process calculus SC, whose processes $P \in \mathcal{P}$ are:

$$P \;::=\; 0 \mid r.P \mid l(P) \mid k_1(P) \mid k_2(P) \mid k_3(P)$$

$$
\begin{array}{ccc}
k_1(X) \xrightarrow{\{l(r.Y)\}}_o k_1(Y) & k_2(X) \xrightarrow{\{l(r.Y)\}}_o k_2(Y) & k_3(X) \xrightarrow{\{l(r.Y)\}}_o k_3(Y) \\
k_1(l(X)) \xrightarrow{\{\diamond r.Y\}}_o k_1(Y) & k_2(l(X)) \xrightarrow{\{\diamond r.Y\}}_o k_2(Y) & k_3(l(X)) \xrightarrow{\{\diamond r.Y\}}_o k_3(Y) \\
 & k_2(l(X)) \xrightarrow{\{r.Y\}}_o k_2(Y) & k_3(X) \xrightarrow{\{l(r.l(Y))\}}_o k_3(l(Y))
\end{array}
$$

Fig. 2. Symbolic transitions for $k_i(X)$ and $k_i(l(X))$

$$
k_2(X) \begin{array}{c}\sim_{\text{strict}}\\ \dot\sim_{\text{large}}\\ \sim_{\text{univ}}\end{array} k_1(X) \begin{array}{c}\not\sim_{\text{strict}}\\ \dot\sim_{\text{large}}\\ \sim_{\text{univ}}\end{array} k_3(X) \begin{array}{c}\not\sim_{\text{strict}}\\ \dot{\not\sim}_{\text{large}}\\ \sim_{\text{univ}}\end{array} k_2(X)
$$

$$
k_2(l(X)) \begin{array}{c}\not\sim_{\text{strict}}\\ \dot{\not\sim}_{\text{large}}\\ \sim_{\text{univ}}\end{array} k_3(l(X))
$$

Fig. 3. Example 1 illustrated

This calculus is not intended to represent a meaningful case study, but just a way to illustrate some peculiarities of our theory. For mnemonic reasons r can be interpreted as a generic resource, $l(\_\,)$ as a locking mechanism, and $k_1(\_\,)$, $k_2(\_\,)$ and $k_3(\_\,)$ as three different access keys which may open and fetch a locked resource. The operational semantics of SC is given by the reduction rules below:

$$
r.P \rightarrow_r P \qquad\qquad k_i(l(r.P)) \rightarrow_o k_i(P) \quad i = 1, 2, 3
$$

The use of a *resource* r is represented by a transition labelled by r, while the use of a key to unlock a process generates a transition labelled by o (*open*).

Let $\mathcal{S}$ be any sound and complete STS whose transitions for the open terms $k_1(X)$, $k_2(X)$, $k_3(X)$, $k_1(l(X))$, $k_2(l(X))$, $k_3(l(X))$ are exactly the ones in Fig. 2 (for instance, they could have been generated by separate specifications provided for each k_i by different system analysts). Observe that in $\mathcal{S}$:

- $k_1(X) \dot\sim_{\text{large}} k_3(X)$, since $k_3(X) \xrightarrow{\{l(r.l(Y))\}}_o k_3(l(Y))$ can be simulated by the instance of $k_1(X) \xrightarrow{\{l(r.Z)\}}_o k_1(Z)$, where Z is replaced by the spatial formula $l(Y)$. Note that, instead, $k_1(X) \not\sim_{\text{strict}} k_3(X)$.
- $k_2(X) \dot\sim_{\text{large}} k_1(X)$, because $k_2(X) \sim_{\text{strict}} k_1(X)$.
- $k_2(X) \dot{\not\sim}_{\text{large}} k_3(X)$ since $k_3(X) \xrightarrow{\{l(r.l(Y))\}}_o k_3(l(Y))$ cannot be simulated via $k_2(X) \xrightarrow{\{l(r.Z)\}}_o k_2(Z)$ with Z replaced by $l(Y)$, as the target processes $k_3(l(Y))$ and $k_2(l(Y))$ are not large bisimilar. In fact, though, in this specific example, the formulae $\diamond r.Y$ and $r.Y$ are satisfied by the same components (i.e., $\{r.p \mid p \in \mathcal{P}\}$), the moves of coordinators $k_3(l(Y))$ and $k_2(l(Y))$, respectively labelled by $\diamond r.Y$ and $r.Y$, cannot be related in the (strict or large) bisimulation game. However it holds that $k_3(l(Y)) \sim_{\text{univ}} k_2(l(Y))$.

The outcome of the above discussion is summarised in Fig. 3. In particular, it shows that $\dot\sim_{\text{large}}$ is non-transitive, i.e., $\dot\sim_{\text{large}} \circ \dot\sim_{\text{large}} \not\subseteq \dot\sim_{\text{large}}$. Moreover, in general, $\sim_{\text{strict}} \subsetneq \dot\sim_{\text{large}} \subsetneq \sim_{\text{univ}}$, i.e., all the inclusions in Theorem 1 are proper.

Therefore, both $\sim_{\text{strict}}$ and $\dot{\sim}_{\text{large}}$ have some pros and cons. In fact $\sim_{\text{strict}}$ is always an equivalence and, in view of an automated verification, its simpler formulation is helpful. Furthermore, being defined as the straightforward notion of bisimilarity on the STS, existing tools and techniques can be reused. On the other hand, $\dot{\sim}_{\text{large}}$ yields a more precise approximation of $\sim_{\text{univ}}$ and, from Theorem 1 it immediately follows that, for sound and complete STSs, $(\dot{\sim}_{\text{large}})^* \Rightarrow \sim_{\text{univ}}$. Hence in using $\dot{\sim}_{\text{large}}$ as a proof technique for $\sim_{\text{univ}}$, transitivity can still be exploited.

Congruence Properties. Call a relation $\cong$ on coordinators an *outer-congruence* if $C[X] \cong D[X]$ implies $C[E[Y]] \cong D[E[Y]]$ for any $E[Y]$, an *inner-congruence* if $C[X] \cong D[X]$ implies $E[C[X]] \cong E[D[X]]$ for any $E[Y]$, and a *quasi-congruence* if it is both an inner- and an outer-congruence. A quasi-congruence which is an equivalence is called a *congruence.* While $\sim_{\text{univ}}$ is an outer-congruence by definition, in general, $\sim_{\text{strict}}$ and $\dot{\sim}_{\text{large}}$ are *not*: taking the calculus SC in Example 1 and the STS $\mathcal{S}$ therein, we have $k_2(X)\dot{\sim}_{\text{large}}k_1(X)$, but $k_2(l(X))\not\dot{\sim}_{\text{large}}k_1(l(X))$. Actually, $k_2(X) \sim_{\text{strict}} k_1(X)$ and thus also $\sim_{\text{strict}}$ is not an outer-congruence. However, since $\sim_{\text{univ}}$ is an outer-congruence and both $\dot{\sim}_{\text{large}}$ and $\sim_{\text{strict}}$ are correct approximations of $\sim_{\text{univ}}$, we can reduce the proof of $C[E[Y]] \sim_{\text{univ}} D[E[Y]]$ to the proof of $C[X]\dot{\sim}_{\text{large}}D[X]$ or $C[X] \sim_{\text{strict}} D[X]$.

Many SOS formats have been introduced to guarantee that bisimilarity is a congruence. This property can be lifted to the symbolic level for PC in De Simone format [17] (a special case of the algebraic format) by taking the STS defined via the Prolog program $\mathit{Prog}_{\mathrm{A}}(\mathrm{SC})$ mentioned in § 2. Moreover, by Definition 4, the absence of spatial operators in the premises of De Simone rules, and hence in the formulae used as labels in the STS, guarantees $\sim_{\text{strict}} = \dot{\sim}_{\text{large}}$.

Proposition 1. *If* PC *is in De Simone format, then* $\mathit{Prog}_{\mathrm{A}}(\mathsf{PC})$ *yields a* STS *where* $\sim_{\text{strict}}$ *is a congruence and* $\sim_{\text{strict}} = \dot{\sim}_{\text{large}}$.

The generalisation of Proposition 1 to other SOS formats (e.g., GSOS) is non-trivial, because they are incomparable w.r.t. the algebraic format and thus $\mathit{Prog}_{\mathrm{A}}(\mathsf{PC})$ (see § 2) cannot be exploited to define a sound and complete STS.

4 Syntactic and Semantic Redundancy

A sound and complete STS may have several different symbolic transitions departing from the same coordinator $C[X]$ but whose instances cover non-disjoint sets of component behaviours. In this section we discuss the influence of redundant symbolic specifications on symbolic bisimilarities. The following example shows that we can distinguish between two kinds of redundancy: syntactic and semantic.

Example 2. Consider the calculus SC in Example 1, where $k_1(X)\dot{\sim}_{\text{large}}k_3(X)$, but $k_1(X) \not\sim_{\text{strict}} k_3(X)$ (see Fig. 3). The equivalence $\sim_{\text{strict}}$ distinguishes the two coordinators because of the symbolic transition $k_3(X) \xrightarrow{\{l(r.l(Y))\}}_{o} k_3(l(Y))$, which is an instance of the more general transition $k_3(X) \xrightarrow{\{l(r.Y)\}}_{o} k_3(Y)$. This is what we call *syntactic* redundancy.

On the other hand, $k_2(l(X)) \not\sim_{\text{strict}} k_3(l(X))$ and $k_2(l(X)) \not\dot{\sim}_{\text{large}} k_3(l(X))$, while $k_2(l(X)) \sim_{\text{univ}} k_3(l(X))$. Roughly, this is due to the fact that the two distinct symbolic transitions $k_2(l(X)) \xrightarrow{r.Y}_o k_2(Y)$ and $k_2(l(X)) \xrightarrow{\diamond r.Y}_o k_2(Y)$ characterise the same set of concrete component transitions (since, in SC, the different formulae $r.Y$ and $\diamond r.Y$ are satisfied by the same processes). This is an aspect of what we call *semantic* redundancy (in general more complex cases can arise, whose solution is not as obvious as here and that cannot be recasted simply in terms of formula equivalences).

4.1 Syntactic Redundancy and Irredundant Bisimilarity

For solving syntactic redundancy the idea is to consider a symbolic bisimulation that takes into account only the "more general" symbolic transitions. For simplicity, we consider calculi without structural axioms.

Definition 5 (Irredundant Transition). *Given a coordinator $C[X]$ in an* STS*, a transition $C[X] \xrightarrow{\varphi}_a C'[Y]$ is called* redundant *if there exists a transition $C[X] \xrightarrow{\psi}_a C''[Z]$ and a spatial formula $\chi \neq Y$ such that $C''[\chi] = C'[Y]$, and $\psi[\chi/Z] = \varphi$. A transition is called* irredundant *if it is not redundant.*

In Example 1, the presence of the (irredundant) transition $k_3(X) \xrightarrow{l(r.Y)}_o k_3(Y)$ makes $k_3(X) \xrightarrow{l(r.l(Y))}_o k_3(l(Y))$ a redundant transition.

Definition 6 ($\dot{\sim}_{\text{irred}}$). *A symmetric relation $\approx$ on coordinators is an* irredundant symbolic bisimulation *if for all $C[X]$, $D[X]$ such that $C[X] \approx D[X]$, for any irredundant transition $C[X] \xrightarrow{\varphi}_a C'[Y]$, there is a transition $D[X] \xrightarrow{\varphi}_a D'[Y]$ such that $C'[Y] \approx D'[Y]$. The largest irredundant symbolic bisimulation $\dot{\sim}_{\text{irred}}$ is called* irredundant symbolic bisimilarity*.*

Like large bisimilarity, also $\dot{\sim}_{\text{irred}}$ might fail to be an equivalence (because of the lack of transitivity). However, the syntactical property in Proposition 2, when satisfied by an STS, guarantees transitivity.

Proposition 2. *Let $\mathcal{S}$ be an* STS *such that for any redundant symbolic transition $C[X] \xrightarrow{\varphi}_a C'[Y]$, if $C[X] \xrightarrow{\psi}_a C''[Z]$ and there exists a spatial formula χ with $\psi[\chi/Z] = \varphi$, then $C''[\chi] = C'[Y]$. Then $\dot{\sim}_{\text{irred}}$ is transitive.*

As mentioned above, large and irredundant bisimilarities, although arising from similar motivations, are (in general) not comparable. To see, for instance, that $\dot{\sim}_{\text{irred}} \not\subseteq \dot{\sim}_{\text{large}}$ consider Example 2. Recall that $k_2(X) \not\dot{\sim}_{\text{large}} k_3(X)$, but instead $k_2(X) \dot{\sim}_{\text{irred}} k_3(X)$, since transition $k_3(X) \xrightarrow{l(r.l(Y))}_o k_3(l(Y))$ is redundant and thus $k_2(X)$ and $k_3(X)$ have the "same" irredundant transitions. An analogous counterexample shows that $\dot{\sim}_{\text{large}} \not\subseteq \dot{\sim}_{\text{irred}}$ (see [6]). Hence it can be useful to combine $\dot{\sim}_{\text{large}}$ and $\dot{\sim}_{\text{irred}}$, as, of course, for any sound and complete STS, $(\dot{\sim}_{\text{large}} \cup \dot{\sim}_{\text{irred}})^* \Rightarrow \sim_{\text{univ}}$. The relationships between the bisimilarities introduced so far are summarised in Fig. 1(a), where arrows represent subset inclusion.

Again, the absence of spatial operators in the premises of De Simone rules, and hence in the formulae used as labels in the STS, guarantees $\sim_{\text{strict}} = \dot{\sim}_{\text{irred}}$.

Proposition 3. *If* PC *is in De Simone format, then* $\mathit{Prog}_{\mathrm{A}}(\mathsf{PC})$ *yields a* STS *where* $\sim_{\mathrm{strict}} = \dot{\sim}_{\mathrm{irred}}$.

4.2 On Semantic Redundancy

The fact that $\dot{\sim}_{\mathrm{large}}$ and $\dot{\sim}_{\mathrm{irred}}$ are incomparable shows that large bisimulation goes beyond syntactic redundancy. Large bisimilarity has been introduced to avoid the distinction between two coordinators $C[X]$ and $D[X]$ that can perform the same transitions, apart from transitions which are, in a sense, instances of other existing transitions. However, in practice, since redundancy check is nested inside the definition, $\dot{\sim}_{\mathrm{large}}$ can deal with a more general notion of redundancy, which has a semantic flavour.

The ideal situation would be when the whole hierarchy in Fig. 1(a) collapses into the simplest symbolic bisimilarity $\sim_{\mathrm{strict}}$, which could then be used as a *complete* proof technique for $\sim_{\mathrm{univ}}$.

However, when sketching the proof of the possible implication $\sim_{\mathrm{univ}} \Rightarrow \sim_{\mathrm{strict}}$, one soon realizes that $\sim_{\mathrm{univ}}$ can hardly be formulated as a strict bisimilarity. In fact assume $C[X] \sim_{\mathrm{univ}} D[X]$, and take any symbolic move $C[X] \xrightarrow{\{\varphi(Y)\}}_a C'[Y]$ of a sound and complete STS. Then, by soundness, we know that $\forall p_i, q_i$ such that $p_i \models \varphi(q_i)$ we have $C[p_i] \rightarrow_a C'[q_i]$. Then, since $C[X] \sim_{\mathrm{univ}} D[X]$, for any such move, we must have $D[p_i] \rightarrow_a d_i$, with $d_i \sim C'[q_i]$. By completeness, it must be the case that there exist $\varphi_i(Z), D'_i[Z], q'_i$ with $D[X] \xrightarrow{\{\varphi_i(Z)\}}_a D'_i[Z]$ such that $p_i \models \varphi_i(q'_i)$ and $D'_i[q'_i] = d_i$, meaning that in general, according to $\sim_{\mathrm{univ}}$, a symbolic move of $C[X]$ can be simulated via the joint effort of several symbolic moves of $D[X]$. More precisely, the choice of the symbolic move $D[X] \xrightarrow{\{\varphi_i(Z)\}}_a D'_i[Z]$ is dependent on the components p_i and q_i that $C[X]$ is going to use. Thus, the difference between the symbolic and the universal approach is essentially the difference between "early" and "late" semantics, based on the time in which p_i and q_i are fixed (before the choice of transition $D[X] \xrightarrow{\{\varphi_i(Z)\}}_a D'_i[Z]$ in $\sim_{\mathrm{univ}}$, after in $\sim_{\mathrm{strict}}$).

The distinction between early and late is inessential provided that either (1) each formula uniquely characterises exactly one p_i and one q_i, or (2) the set of processes satisfying any two different formulae are disjoint and all symbolic transitions with the same source have different labels. Only having the calculus at hand, these semantic assumptions can be verified and eventually exploited. Finding a general way to face this issue is a challenging open problem.

The discussion about semantic redundancy also suggests that syntactical formats are not enough for guaranteeing that exact approximations of $\sim_{\mathrm{univ}}$ can be inferred. Indeed, the next example shows that even De Simone format cannot ensure that $\sim_{\mathrm{strict}} = \sim_{\mathrm{univ}}$.

Example 3. Let us extend finite CCS with the operators $\mathit{one}_a(\_)$, $\mathit{stop}(\_)$, and with the SOS rule

$$\frac{P \rightarrow_\mu Q}{\mathit{one}_a(P) \rightarrow_a \mathit{stop}(Q)}$$

The resulting calculus CCS* adheres to the De Simone format. One can easily verify that the processes $C[X] = a.0 + a.b.0 + a.\mathit{one}_b(X)$ and $D[X] = a.0 + a.b.0 +$

$stop(X)$ are universally bisimilar, but in the STS generated by $Prog_{\mathrm{A}}(\text{CCS}^*)$ (see § 2), they are not strict bisimilar (intuitively, because instantiation is dynamic in the symbolic bisimulation game, while it is decided once and forever for $\sim_{\mathrm{univ}}$).

5 Symbolic Trace Semantics

Bisimilarity relates states that have "the same" branching structure. Often this feature is not directly relevant to the abstract view of the system, provided that the states can perform "the same" sequences of transitions. To this purpose, *trace semantics*—following the terminology introduced in [23]—are sometimes preferred to bisimilarity. In this section we define a hierarchy of symbolic trace semantics.

A variety of different (decorated) trace semantics has been studied in the literature (e.g., ready traces, failure traces, completed traces, accepting traces, see [2] for an overview), each relying on particular interleaved sequences of actions and state predicates. Here we just consider the basic case of finite traces (hereafter simply called traces), where finite sequences of actions are observed.

Given a component $p \in \mathcal{P}$, a *trace* of p is a finite sequence $\varsigma = a_1 a_2 \cdots a_n \in \Lambda^*$ such that there exist n components $p_1, \ldots, p_n$ with $p {\rightarrow_{a_1}} p_1 {\rightarrow_{a_2}} \cdots {\rightarrow_{a_n}} p_n$ (abbreviated $p {\rightarrow_{\varsigma}} p_n$ or just $p {\rightarrow_{\varsigma}}$). The *trace language* of p is the set $\mathbf{L}(p) = \{\varsigma \in \Lambda^* \mid p {\rightarrow_{\varsigma}}\}$. Two components p and q are *trace equivalent*, written $p \simeq q$, if $\mathbf{L}(p) = \mathbf{L}(q)$. Quite obviously, for all components $p, q \in \mathcal{P}$, if $p \sim q$ then $p \simeq q$ (but the converse implication does not hold in general). As in the case of bisimilarity, the natural way of lifting trace equivalence to coordinators consists of comparing all their closed instances, defining $C[X] \simeq_{\mathrm{univ}} D[X]$ iff for all $p \in \mathcal{P}$, $C[p] \simeq D[p]$.

A different notion of trace equivalence for coordinators is readily obtained if an STS for the calculus is available. In fact, symbolic traces can be straightforwardly defined as sequences of (formula,action)-pairs.

Definition 7 ($\simeq_{\mathrm{strict}}$). *A* symbolic trace *of a coordinator $C[X]$ in an* STS $\mathcal{S}$ *is a finite sequence $\zeta = \langle\varphi_1, a_1\rangle\langle\varphi_2, a_2\rangle \cdots \langle\varphi_m, a_m\rangle \in (\Phi \times \Lambda)^*$, where Φ is the set of formulae in L_{PC}, such that there exist m coordinators $C_1[X_1], \ldots, C_m[X_m]$ with $C[X] \mathrel{-\{\varphi_1\}\!\!\rightarrow}_{a_1} C_1[X_1] \mathrel{-\{\varphi_2\}\!\!\rightarrow}_{a_2} \cdots \mathrel{-\{\varphi_m\}\!\!\rightarrow}_{a_m} C_m[X_m]$, (abbreviated $C[X] {\rightarrow_{\zeta}} C_m[X_m]$ or just $C[X] {\rightarrow_{\zeta}}$). The* strict trace language *of $C[X]$ is the set $\mathbf{L}(C[X]) = \{\zeta \in (\Phi \times \Lambda)^* \mid C[X] {\rightarrow_{\zeta}}\}$. Two coordinators $C[X]$ and $D[X]$ are* strict trace equivalent*, written $C[X] \simeq_{\mathrm{strict}} D[X]$, if $\mathbf{L}(C[X]) = \mathbf{L}(D[X])$.*

We have $\sim_{\mathrm{strict}} \Rightarrow \simeq_{\mathrm{strict}}$ (see Theorem 2 at the end of the current section), and the inclusion is proper, as shown by the next example.

Example 4. Consider the calculus SCM, a restriction-free version of the ambient calculus [11] with asynchronous CCS-like communication, whose set of processes $\mathcal{P}$ is defined in Fig. 4. The parallel operator $|$ is associative and commutative, with 0 the identity, $a, b, \ldots$ are channels and $n, m, \ldots$ ambient names. The operational semantics of SCM, defined by SOS rules, states that, in the ambient calculus

$$P \ ::= \ 0 \mid \bar{a} \mid a.P \mid \mathit{open}\ n.P \mid \mathit{in}\ n.P \mid \mathit{out}\ n.P \mid n[P] \mid P|P$$

$$\varphi ::= X \mid \diamond.\varphi \mid 0 \mid \alpha.\varphi \mid n[\varphi] \mid \varphi_1|\varphi_2$$

$$\frac{}{n[P] \mid \mathit{open}\ n.Q \ \rightarrow \ P|Q}\ (\mathit{open}) \qquad \frac{}{n[P]|m[\mathit{in}\ n.Q|R] \ \rightarrow \ n[P|m[Q|R]]}\ (\mathit{in})$$

$$\frac{}{n[P|m[\mathit{out}\ n.Q|R]] \ \rightarrow \ n[P]|m[Q|R]}\ (\mathit{out}) \qquad \frac{}{n[a.P|\bar{a}|Q] \ \rightarrow \ n[P|Q]}\ (\mathit{comm})$$

$$\frac{P \ \rightarrow \ Q}{P|R \ \rightarrow \ Q|R}\ (\mathit{par}) \qquad \frac{P \ \rightarrow \ Q}{n[P] \ \rightarrow \ n[Q]}\ (\mathit{amb})$$

Fig. 4. Syntax, associated logic and operational semantics of SCM

style, processes can move in, move out and open environments, and also asynchronously communicate within them. Being $\Lambda = \{\tau\}$, transition labels are not relevant and are omitted, i.e., we write $\rightarrow$ and $\mathrel{-\{\varphi\}\!\!\rightarrow}$ in place of $\rightarrow_\tau$ and $\mathrel{-\{\varphi\}\!\!\rightarrow}_\tau$. Figure 4 also shows the formulae φ of the associated logic $\mathsf{L}_{\mathrm{SCM}}$, where X is a process variable, n an ambient name and $\alpha \in \{a, \bar{a}, \mathit{open}\ n, \mathit{in}\ n, \mathit{out}\ n\}$. Since transitions are unlabelled, the modal operator does not mention any action.

Let us consider $C[X] = m[a.(a.0|b.0)|X]$ and $D[X] = m[a.0|a.b.0|X]$, and the STS generated by $\mathit{Prog}_{\mathrm{A}}(\mathrm{SCM})$ (see § 2). It holds $C[X] \not\simeq_{\mathrm{strict}} D[X]$, since, for instance, the transition $C[X] \mathrel{-\{\bar{a}|Y\}\!\!\rightarrow} C'[Y] = m[a.0|b.0|Y]$ could only be simulated by the transition $D[X] \mathrel{-\{\bar{a}|Y\}\!\!\rightarrow} D'[Y] = m[a.b.0|Y]$, but $C'[Y] \not\simeq_{\mathrm{strict}} D'[Y]$ (since $D'[X]$ can not simulate $C'[X] \mathrel{-\{\bar{b}|Z\}\!\!\rightarrow} m[a.0|Z]$). On the other hand, $C[X] \simeq_{\mathrm{strict}} D[X]$. In fact, either $C[X] \mathrel{-\{\bar{a}|Y_1\}\!\!\rightarrow} C_1[Y_1] \mathrel{-\{\bar{a}|Y_2\}\!\!\rightarrow} C_2[Y_2] \mathrel{-\{\bar{b}|Y_3\}\!\!\rightarrow} m[Y_3]$ or $C[X] \mathrel{-\{\bar{a}|Y_1\}\!\!\rightarrow} C_1[Y_1] \mathrel{-\{\bar{b}|Y_2\}\!\!\rightarrow} C_3[Y_2] \mathrel{-\{\bar{a}|Y_3\}\!\!\rightarrow} m[Y_3]$, for obvious $C_1[Y_1], C_2[Y_2]$ and $C_3[Y_2]$, and hence, missing the label components, the language $\mathbf{L}(C[X])$ is

$$\{\lambda\ ,\ \bar{a}|Y_1\ ,\ \bar{a}|Y_1\ \bar{a}|Y_2\ ,\ \bar{a}|Y_1\ \bar{b}|Y_2\ \} \cup \{\bar{a}|Y_1\ \bar{a}|Y_2\ \bar{b}|Y_3\ ,\ \bar{a}|Y_1\ \bar{b}|Y_2\ \bar{a}|Y_3\} \cdot \mathbf{L}(m[Y_3]),$$

where "$\cdot$" is language concatenation and λ is the empty trace. The language $\mathbf{L}(D[X])$ is the same as $\mathbf{L}(C[X])$.

An alternative notion of symbolic trace can be introduced by noting that formulae φ and ψ labelling two consecutive transitions can be composed by replacing the variable occurring in φ with the formula ψ.

Definition 8 (Tight Traces). *Let $\zeta = \langle\varphi_1, a_1\rangle\langle\varphi_2, a_2\rangle \cdots \langle\varphi_m, a_m\rangle \in (\Phi \times \Lambda)^*$ be a symbolic trace of a coordinator $C[X]$. The corresponding* tight trace *is the pair $\mathit{comp}(\zeta) = (\varphi, a_1 a_2 \cdots a_m) \in \Phi \times \Lambda^*$, where $\varphi = \varphi_1[\varphi_2[\ldots[\varphi_m/X_{m-1}]\ldots/X_2]/X_1]$.*

Tight traces can now be used to better approximate $\simeq_{\mathrm{univ}}$.

Definition 9 ($\simeq_{\mathrm{stight}}$). *The* strict tight trace language *of $C[X]$ is $\mathbf{C}(C[X]) = \{\rho \in \Phi \times \Lambda^* \mid \exists\zeta \in \mathbf{L}(C[X]).\ \rho = \mathit{comp}(\zeta)\}$. Two coordinators $C[X]$ and $D[X]$ are* strict tight trace equivalent*, written $C[X] \simeq_{\mathrm{stight}} D[X]$, if $\mathbf{C}(C[X]) = \mathbf{C}(D[X])$.*

Since *comp*(.) is a function, $\simeq_{\text{strict}} \Rightarrow \simeq_{\text{stight}}$ (Theorem 2). As shown by the next example the inclusion is proper (since different symbolic traces can give rise to the same tight trace).

Example 5. Consider the calculus FOO, defined over the unsorted signature $\Sigma = \{a, f(.), g(.), h(.), l(.), k(.)\}$, with the rules:

$$\begin{array}{cc} f(h(X)) \rightarrow_a h(X) & g(l(X)) \rightarrow_a l(X) \\ h(X) \rightarrow_b X & l(h(X)) \rightarrow_b X \\ f(X) \rightarrow_a k(X) & g(l(h(X))) \rightarrow_a k(X) \end{array}$$

From the symbolic transitions below, generated by $Prog_{\text{A}}$(FOO), it is easy to see that $f(X) \not\simeq_{\text{strict}} g(l(X))$, while $f(X) \simeq_{\text{stight}} g(l(X))$ (the traces $\langle h(Y), a\rangle\langle Z, b\rangle$ and $\langle Y, a\rangle\langle h(Z), b\rangle$ collapse in the tight trace $\langle h(Z), a\ b\rangle$).

$$\begin{array}{lllll} f(X) & -\{h(Y)\}\!\succ_a h(Y) & -\{Z\}\!\succ_b Z \ldots & \qquad & f(X) -\{Y\}\!\succ_a k(Y) \\ g(l(X)) & -\{Y\}\!\succ_a l(Y) & -\{h(Z)\}\!\succ_b Z \ldots & \qquad & g(l(X)) -\{h(Y)\}\!\succ_a k(Y) \end{array}$$

As it happens for bisimilarity, the requirement of exact match between the formulae observed in a trace can be relaxed for spatial formulae.

Definition 10 (Saturated Trace). *A* saturated trace *of* $C_1[X_1]$ *is a finite sequence* $\zeta = \langle\varphi_1, a_1\rangle\langle\varphi_2, a_2\rangle \cdots \langle\varphi_m, a_m\rangle \in (\Phi \times \Lambda)^*$*, such that there exist* m *coordinators* $C_2[X_2], ..., C_{m+1}[X_{m+1}]$ *and* m *spatial formulae* $\psi_1, ..., \psi_m$ *with:*

1. $C_i[X_i] -\{\varphi'_i\}\!\succ_{a_i} C'_i[Y_i]$,
2. $C_i[X_i] = C'_{i-1}[\psi_{i-1}]$,
3. $\varphi_i = \varphi'_i[\psi_i/Y_i]$,

for all $i \in [1, m]$*. The* saturated trace language *of* $C[X]$ *is the set* $\mathbf{S}(C[X])$ *of its saturated traces.*

A saturated trace in $\mathcal{S}$ is basically a symbolic trace in the STS obtained from $\mathcal{S}$ by adding for each symbolic transition $C[X] -\{\varphi\}\!\succ_a C'[Y]$ all of its instances, i.e., a transition $C[X] -\{\varphi[\psi/Y]\}\!\succ_a C'[\psi]$ for any spatial formula ψ.

Definition 11 ($\dot{\simeq}_{\text{large}}$). $C[X]$ *and* $D[X]$ *are* large trace pre-equivalent*, written* $C[X] \dot{\simeq}_{\text{large}} D[X]$ *if* $\mathbf{L}(C[X]) \subseteq \mathbf{S}(D[X])$ *and* $\mathbf{L}(D[X]) \subseteq \mathbf{S}(C[X])$*.* Large trace equivalence $\simeq_{\text{large}}$ *is the transitive closure of* $\dot{\simeq}_{\text{large}}$*.*

Analogously to large bisimilarity, large trace pre-equivalence $\dot{\simeq}_{\text{large}}$ might not be transitive. Hence its transitive closure is considered to obtain an equivalence.

Finally, to overcome syntactic redundancy, a notion of symbolic trace equivalence can be defined exploiting irredundant transition (see Definition 5).

Definition 12 ($\dot{\simeq}_{\text{irred}}$). *Let* $\mathbf{I}(C[X])$ *be the subset of* $\mathbf{L}(C[X])$ *containing traces composed by irredundant transitions only. Two coordinators* $C[X]$ *and* $D[X]$ *are* irredundant trace pre-equivalent*, written* $C[X] \dot{\simeq}_{\text{irred}} D[X]$ *if* $\mathbf{I}(C[X]) \subseteq \mathbf{L}(D[X])$ *and* $\mathbf{I}(D[X]) \subseteq \mathbf{L}(C[X])$*.* Irredundant trace equivalence $\simeq_{\text{irred}}$ *is the transitive closure of* $\dot{\simeq}_{\text{irred}}$*.*

The "tight" versions of $\dot{\simeq}_{\mathrm{large}}$ and $\dot{\simeq}_{\mathrm{irred}}$ can be defined as in the case of the strict equivalence, by resorting to the corresponding tight trace languages (see Definition 8). This leads to the trace pre-equivalences $\dot{\simeq}_{\mathrm{ltight}}$ and $\dot{\simeq}_{\mathrm{itight}}$ refined by the original ones, i.e., such that $\dot{\simeq}_{\mathrm{large}} \subseteq \dot{\simeq}_{\mathrm{ltight}}$ and $\dot{\simeq}_{\mathrm{irred}} \subseteq \dot{\simeq}_{\mathrm{itight}}$.

The theorem below clarifies all the relationships between all bisimilarities and trace semantics introduced so far. As expected, each kind of bisimilarity is finer than the corresponding trace semantics. The diamond involving the strict, large, irredundant and universal relations also holds for trace semantics.

Theorem 2. *Given any sound and complete* STS *for a process calculus PC, the relationships indicated in Fig. 1(b) hold, where arrows represent subset inclusion.*

6 Concluding Remarks

We introduced in [5] a methodology for reasoning about the operational and abstract semantics of open systems, viewed as coordinators in suitable process calculi, with focus on bisimilarity. Here, we have analysed how redundancy in STS may influence the quality of the approximation of universal bisimilarity $\sim_{\mathrm{univ}}$, and we have provided a hierarchy of symbolic bisimilarities (Fig. 1(a)), where alternative equivalences for approximating $\sim_{\mathrm{univ}}$ are proposed and studied. The approach has been extended to non-branching semantics, and, correspondingly, a hierarchy of symbolic equivalences (Fig. 1(b)) has been established.

As a matter of future investigation, we plan to develop the treatment of names and name restriction in order to deal with open systems where fresh or secret resources are a main concern. In particular, the notion of STS and the underlying process logic should be extended to deal with a logical notion of freshness, possibly taking inspiration from [9, 10]. The higher-order unification mechanism of λ-Prolog [28] could provide a convenient framework for the construction of the STS.

In the setting of name-based calculi, the openness of a system can involve not only process variables, but also communication on shared channels. This view is not in conflict with our approach, but it rather suggests an appealing direction for the future development of our work. Also for value-passing and name-based calculi, transition labels are typically structured. As shown in [22], this fact can be profitably used in a symbolic approach to define tractable behavioural equivalences, and, although labels can always be seen as a plain set, we plan to extend our symbolic approach to cope with structured transition labels.

Symbolic equivalences are intended as means for providing tractable approximations of corresponding equivalences defined by universal quantification over the set of components. Hence, on the applicative side, we expect some outcomes in the direction of the automated verification of open systems. Specifically, we are developing software tools, which, exploiting the Prolog program associated to an SOS specification, support the automated verification of symbolic equivalences. A first prototype tool, called SEA (Symbolic Equivalence Analyzer), has been developed in [30].

Pursuing further the Prolog-based algorithmic construction of STSs, we plan to investigate the use of *meta* Logic Programming for the programmable definition of transitions, and thus more specific (automated) reasoning over the structure of a PC. Moreover, also *abductive* Logic Programming is worth being considered for hypothetical, assumption-based reasoning about formulae, e.g., "under which assumptions the process $P \mid X$ can evolve so as to satisfy a given property?", which is typically relevant in open and dynamic system engineering [4, 3].

Related Work. The notion of STS has been influenced by several related formalisms. Symbolic approaches to behavioural equivalences can be found in [22, 34], while the idea of using spatial logic formulae as an elegant mathematical tool for combining structural and behavioural constraints has been separately proposed in [12, 18]. Many different kinds of labelled transition systems for coordinators have been previously proposed in the literature (e.g., structured transition systems [15], context systems [25], tile logic [20], conditional transition systems [32]). Roughly, the distinguishing feature of our approach is the greater generality of symbolic transition labels which account for spatial constraints over unspecified components.

In case of LTSs with a unique label τ (that can be regarded as reduction semantics), our approach seems to share some analogies with narrowing techniques used in rewrite systems, and it would be interesting to formally compare the two approaches. For a CCS fragment, early studies about terms with variables [21, 27] have shown that the presence of symbolic actions can be helpful in proving the completeness of axioms for bisimilarity. This work could be inspiring for addressing analogous issues in other calculi.

Some close relations exist also with the work on *modal transition systems* [24], where both transitions that *must* be performed and transitions which are only *possible* can be specified. Consequently the syntax of the calculus is extended with two kind of prefix operators $\Box a.()$ and $\Diamond a.()$. We recall also the logical process calculus of [14], which mixes CCS and a form of μ-calculus, to allow the logical specification of some components of the system. Our process logic exhibits some similarities both with the calculus underlying modal transition systems and with the logical process calculus. However, the purpose of the mentioned formalisms is to provide a loose specification of a system, where some components are characterised by means of logical formulae. Instead, in our case open systems are modelled within the original calculus and the STS fully describe their semantics by using the logic to characterise synthetically their possible transitions.

Process calculi have been traditionally used for cryptographic protocol verification, exploiting symbolic semantics for dealing with the infiniteness of the attacker models (typically due to the unconstrained generative power of intruders), see e.g. [1, 13] and especially the unification-based approach [7]. Such similarities suggest possible applications of our framework to security-oriented calculi.

The problem of the universal quantification over components in the definition of behavioural equivalences for open systems has its dual counterpart in the contextual closure needed when the bisimilarity on components $\sim$ is not a congruence and one defines the largest congruence $\simeq$ contained in $\sim$, by letting $p \simeq q$ if for all contexts $C[.]$, $C[p] \sim C[q]$. To avoid universal quantification on contexts, several authors (see [36, 26, 35]) propose a symbolic transition system for components whose labels are the "minimal" contexts needed by the component in order to evolve. Understanding to which extent this duality can be pursued and exploited is an interesting direction for future research.

References

1. M. Abadi and M.P. Fiore. Computing symbolic models for verifying cryptographic protocols. *Proc. 14th IEEE Computer Security Foundations Workshop*, pp. 160–173. IEEE Computer Society Press, 2001.
2. L. Aceto, W.J. Fokkink, and C. Verhoef. Structural operational semantics. *Handbook of Process Algebra*, pp. 197–292. Elsevier Science, 2001.
3. R. Allen and D. Garlan. A formal basis for architectural connectors. *ACM Transactions on Software Engineering and Methodology*, 3(6):213–249, 1997.
4. L.F. Andrade, J.L. Fiadeiro, L. Gouveia, G. Koutsoukos, and M Wermelinger. Coordination for orchestration. *Proc. COORDINATION 2002*, *LNCS* 2315, pp. 5–13. Springer, 2002.
5. P. Baldan, A. Bracciali, and R. Bruni. Bisimulation by unification. *Proc. AMAST 2002*, *LNCS* 2422, pp. 254–270, Springer 2002.
6. P. Baldan, A. Bracciali, and R. Bruni. Symbolic equivalences for open systems. Technical Report TR-03-16, Department of Computer Science, University of Pisa, 2003.
7. M. Boreale. Symbolic trace analysis of cryptographic protocols. *Proc. ICALP'01*, *LNCS* 2076, pp. 667–681. Springer, 2001.
8. L. Caires. *A Model for Declarative Programming and Specification with Concurrency and Mobility.* PhD thesis, Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 1999.
9. L. Caires and L. Cardelli. A spatial logic for concurrency (part I). In *Proc. TACS 2001*, *LNCS* 2215, pp. 1–37. Springer, 2001.
10. L. Caires and L. Cardelli. A spatial logic for concurrency (part II). In *Proc. CONCUR 2002*, *LNCS* 2421, pp. 209–225. Springer, 2002.
11. L. Cardelli and A.D. Gordon. Mobile ambients. *Proc. FoSSaCS'98*, *LNCS* 1378, pp. 140–155. Springer, 1998.
12. L. Cardelli and A.D. Gordon. Anytime, anywhere. modal logics for mobile ambients. In *Proc. POPL 2000*, pp. 365–377. ACM, 2000.
13. E.M. Clarke, S. Jha, and W. Marrero. Using state space exploration and a natural deduction style message derivation engine to verify security protocols. In *Proc. PROCOMET'98*, Chapmann & Hall, 1998.
14. R. Cleaveland and G. Lüttgen. A logical process calculus. In *ENTCS*, 2002.
15. A. Corradini and U. Montanari. An algebraic semantics for structured transition systems and its application to logic programs. *Theoret. Comput. Sci.*, 103:51–106, 1992.
16. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.
17. R. De Simone. Higher level synchronizing devices in MEIJE–SCCS. *Theoret. Comput. Sci.*, 37:245–267, 1985.

18. J.L. Fiadeiro, T. Maibaum, N. Martí-Oliet, J. Meseguer, and I. Pita. Towards a verification logic for rewriting logic. *Proc. WADT'99*, *LNCS* 1827, pp. 438–458. Springer, 2000.
19. R. Focardi and R. Gorrieri Classification of Security Properties (Part I: Information Flow) FOSAD'01 - Tutorial Lectures, *LNCS* 2171, pp. 331–396. Springer, 2001.
20. F. Gadducci and U. Montanari. The tile model. *Proof, Language and Interaction: Essays in Honour of Robin Milner*, pp. 133–166. MIT Press, 2000.
21. R. van Glabbeek. A complete axiomatization for branching bisimulation congruence of finite-state behaviours. In *Proc. MFCS'93*, *LNCS* 711, pp 473-484, Springer, 1993.
22. M. Hennessy and H. Lin. Symbolic bisimulations. *Theoret. Comput. Sci.*, 138:353–389, 1995.
23. C.A.R. Hoare. A model for communicating sequential processes. *On the Construction of Programs*. Cambridge University Press, 1980.
24. K. G. Larsen and B. Thomsen. A modal process logic. In *Proceedings of LICS*, pages 203–210. IEEE, 1988.
25. K.G. Larsen and L. Xinxin. Compositionality through an operational semantics of contexts. *Proc. ICALP'90*, *LNCS* 443, pp. 526–539. Springer, 1990.
26. J.J. Leifer and R. Milner. Deriving bisimulation congruences for reactive systems. *Proc. CONCUR 2000*, *LNCS 1877*, pp. 243–258. Springer, 2000.
27. R. Milner. A complete axiomatisation for observational congruence of finite-state behaviours. *Information and Computation*, 81:227–247, 1989.
28. D. Miller and G. Nadathur. Higher-order logic programming. *Handbook of Logics for Artificial Intelligence and Logic Programming*, volume 5, pp. 499–590. Clarendon Press, 1998.
29. R. Milner, J. Parrow, and J. Walker. A calculus of mobile processes, I and II. *Inform. and Comput.*, 100(1):1–40,41–77, 1992.
30. R. Nunziato. Sviluppo dell'applicazione SEA per la verifica di sistemi aperti. Master Thesis, Department of Computer Science, University of Pisa, 2003. (In Italian.)
31. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981.
32. A. Rensink. Bisimilarity of open terms. *Inform. and Comput.*, 156(1-2):345–385, 2000.
33. D. Sangiorgi. *Expressing Mobility in Process Algebras: First-Order and Higher-Order Paradigms*. PhD thesis, LFCS, University of Edinburgh, 1993. CST-99-93 (also published as ECS-LFCS-93-266).
34. D. Sangiorgi. A theory of bisimulation for the π-calculus. *Acta Inform.*, 33:69–97, 1996.
35. V. Sassone and P. Sobocinski. Deriving bisimulation congruences using 2-categories. In *Nordic Journal of Computing*, volume 10. Elsevier, 2002.
36. P. Sewell. From rewrite rules to bisimulation congruences. *Proc. CONCUR'98*, *LNCS* 1466, pp. 269–284. Springer, 1998.

Specifying and Verifying UML Activity Diagrams Via Graph Transformation*

Paolo Baldan[1], Andrea Corradini[2], and Fabio Gadducci[2]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy
[2] Dipartimento di Informatica, Università di Pisa, Italy

Abstract. We propose a methodology for system specification and verification based on UML diagrams and interpreted in terms of graphs and graph transformations. Once a system is modeled in this framework, a temporal graph logic can be used to express some of its relevant behavioral properties. Then, under certain constraints, such properties can be checked automatically. The approach is illustrated over a simple case study, the so-called Airport Case Study, which has been widely used along the first two years of the AGILE GC project.

1 Introduction

The use of visual modeling techniques, like the UML [22], for the design and development of large applications is nowadays well established. In these approaches a system specification consists of several related diagrams, that represent both the statics and the dynamics of the system. Since the development process is made easier if it is possible to reason about the system under development at an early stage, in this paper we sketch a methodology which allows to express interesting behavioral properties of the system in a suitable logic, and, under certain constraints, to verify them automatically. We present our approach by applying it to a simple case study, which has been widely used along the first two years of the AGILE GC project [1], namely the Airport Case Study [2].

The first step of our methodology consists of representing a UML specification as a Graph Transformation System (GTS). Since the various kinds of diagrams used in a UML specification essentially are graphs annotated in various ways, it comes as no surprise that many contributions in the literature use techniques based on the theory of graph transformation to provide an operational semantics for UML behavioral diagrams (see, among others, [10–13, 17, 18]). We will stick to a tiny fragment of the UML, including (suitably restricted) class and instance diagrams for the statics, and activity diagrams for the dynamics of a system specification. The class diagram determines the shape of the graphs that will be used for modeling instance diagrams, called *instance graphs*, while each activity in a behavioral diagram will be represented as a graph transformation rule, describing the effect of such activity on the instance graph.

* Research partially supported by the EU FET – GC Project IST-2001-32747 AGILE and the EC RTN 2-2001-00346 SegraVis.

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 18–33, 2004.

Next, following an approach to the verification of graph transformation systems developed during the last few years (see [3–5]), we shall introduce a temporal logic which allows for formulating relevant properties of a GTS. The logic $\mu\mathcal{L}2$ proposed in [5] (that we slightly modify in order to deal with a more general class of graphs) is a propositional μ-calculus where the basic predicates are monadic second-order formulae interpreted over graphs. For (fragments of) this logic, verification techniques have been proposed for finite and infinite-state systems, which exploit finite approximations of the unfolding of the GTS [4, 5].

The expressiveness of the proposed logic is tested against a collection of properties concerned with the Airport Case Study, which were collected by the members of the AGILE project in a meeting dedicated to modal and temporal logics (the proceedings are available at [1]). Even if monadic second-order logic is very expressive as far as graph properties are concerned, it turns out that some interesting dynamic properties of the Airport Case Study cannot be encoded directly in the proposed logic. Being the logic propositional at the temporal layer, there is no way to write formulae predicating about the properties of a specific object at different times. We briefly outline a more expressive graph logic, which, extending $\mu\mathcal{L}2$ with non-propositional features, allows for overcoming the mentioned limitations. Unfortunately, the verification techniques in [4, 5] do not directly apply to this logic, but we are confident that they can be suitably generalized, at least in the finite-state case.

In the next section, after a brief introduction to the Airport Case Study and to a partial specification of it using UML, we first show how the states of the case study can be represented by graphs, and the corresponding activities by graph transformation rules. Next we introduce the temporal logic for GTSs, and finally we discuss to what extent some relevant properties can be expressed in that logic, and which extensions of such logic would be needed.

2 The Airport Case Study

As anticipated above, in this paper we shall illustrate the main concepts using as running example a fragment of the Airport Case Study, described in [2].

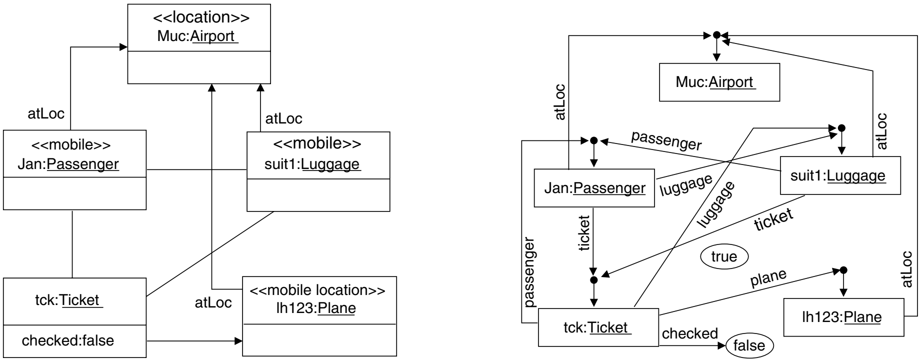
The case study consists of a system representing planes landing and taking off from airports. The planes transport passengers. Departing passengers check in and board the plane; their luggage is loaded in the plane. The plane is ready to take off after all passengers have boarded the plane and their luggage is loaded. After the plane has reached its destination airport, passengers get off the plane and claim their luggage. On board, passengers may perform some activities, such as consuming a meal. The specification and modeling of several aspects of this case study using a variety of formalisms (UML, CommUnity, KLAIM and Graph Transformations) is presented in [2].

We shall consider here only the part of the case study related to the *Departure Use Case*, including the check-in and boarding of passengers, the loading of their luggage and the take-off of the plane. A UML instance diagram representing the initial state of the system is shown in Figure 1 (a), adopting the stereotypes *mobile* and *location* proposed in a recent extension for mobility of the language developed inside the AGILE project; instead, Figure 2 shows an activity diagram describing the relationships among the relevant activities. In the next subsections we discuss how to model this system using graph transformation, representing its states as graphs and the activities as rules.

2.1 Representing States as Hypergraphs

In order to model a UML specification as a graph transformation system, an obvious pre-requisite is the formal definition of the structure of the graphs which represent the states of the system, namely the *instance graphs*. However, there is no common agreement about this: we shall present a novel formalization, which shares some features with the one proposed in [14].

An *instance graph* includes a set of *nodes*, which represent all data belonging to the state of an execution. Some of them represent the elements of primitive data types, while others denote instances of classes. Every node may have at most one outgoing *hyperedge*, i.e., an edge connecting it to zero or more nodes.[1] Conceptually, the node can be interpreted as the "identity" of a data element, while the associated hyperedge, if there is one, contains the relevant information about its state. A node without outgoing hyperedges is a *variable*: variables only appear in transformation rules, never in actual states.



(a) (b)

Fig. 1. An instance diagram (a) and the corresponding instance graph (b)

Typically, an instance of a class C is represented by a node n and by a hyperedge labeled with the pair $\langle \mathit{instanceName} : C \rangle$. This hyperedge has node n as its only *source*, and for each attribute of the class C it has a link (a *target tentacle*) labeled by the name of the attribute and pointing to the node representing the attribute value. For the logic presented later in Section 3, we assume that the *source tentacle* (linking a hyperedge to its source node) is implicitly labeled by

[1] Formally, these graphs are *term graphs* [21].

self. Every instance graph also includes, as unary hyperedges (i.e., hyperedges having only the self tentacle), all constant elements of primitive data types, like integers (0, 1, -1, ...) and booleans (true and false), as well as one edge null:C for each relevant class C.

Figure 1 (a) shows an instance diagram which represents the initial state of the Airport Scenario. As usual, the attributes of an instance may be represented as directed edges labeled by the attribute name, and pointing to the attribute value. The edge is unlabeled if the attribute name coincides with the class of the value (e.g., lh123 is the value of the plane attribute of tck). An undirected edge represents two directed edges between its extremes. The diagram conforms to a class diagram that is not depicted here.

Figure 1 (b) shows the instance graph (according to the above definitions) encoding the instance diagram. Notice that the graph contains two elements of a basic data type, true and false: these are depicted as ovals, which stands actually for a node attached through the self tentacle to a unary hyperedge. Up to a certain extent (disregarding OCL formulas and cardinality constraints), a class diagram can be encoded in a corresponding *class graph* as well; then the existence of a graph morphism (i.e., a structure preserving mapping) from the instance graph to the class graph formalizes the relation of conformance.

In the following we shall depict the states of the system as instance diagrams, which are easier to draw and to understand, but they are intended to represent the corresponding instance graphs.

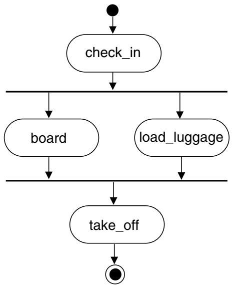


Fig. 2. The Activity Diagram of the Use Case Departure

2.2 Representing Activities as Graph Transformation Rules

Figure 2 shows the activity diagram of the Use Case Departure of the Airport Case Study. This behavioural diagram ignores the structure of the states and the information about which instances are involved in each activity, but stresses the causal dependencies among activities and the possible parallelism among them. More precisely, from the diagram one infers the requirement that board and load_luggage can happen in any order, after check_in and before take_off.

By making explicit the roles of the various instances in the activities, we shall implement each activity as a graph transformation rule. Such rules describe local modifications of the instance graphs resulting from the corresponding activities.

We will show that they provide a correct implementation of the activity diagram, in the sense that the causality and independence relations between the rules are exactly those prescribed in the activity diagram.

Let us first consider the activity board. Conceptually, in the simplified model we are considering, its effect is just to change the location of the passenger (i.e., its atLoc attribute) from the airport to the plane. In the rule which implements the activity, we make explicit the preconditions for its application: 1) the passenger must have a ticket for the flight using that plane; 2) the value of the checked attribute of the ticket must be true; 3) the plane and the passenger must be at the *same* location, which is an airport.

All the above requirements are represented in the graph transformation rule implementing the activity board, shown in Figure 3. Formally, this is a *double-pushout graph transformation rule* [7], having the form $L \xleftarrow{l} K \xrightarrow{r} R$, where L, K and R are instance graphs, and l and r are graph morphisms. In this case l and r are actually inclusions, represented implicitly by the position of nodes and edges in the source and target graphs.

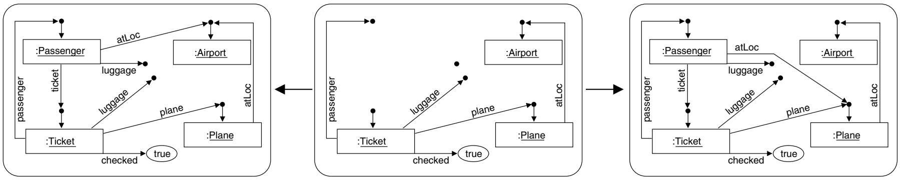


Fig. 3. The graph transformation rule for boarding

Intuitively, a rule states that whenever we find an occurrence of the *left-hand side* L in a graph G we may replace it with the *right-hand side* R. The *interface* graph K and the two morphisms l and r provide the *embedding information*, that is, they specify where R should be glued with the *context* graph obtained from G by removing L. More precisely, an *occurrence* of L in G is a graph morphism $g : L \to G$. The context graph D is obtained by deleting from G all the nodes and edges in $g(L - l(K))$ (thus all the items in the interface K are preserved by the transformation). The insertion of R in D is obtained by taking their disjoint union, and then by identifying for each node or edge x in K its images $g(x)$ in G and $r(x)$ in R: formally, this operation is a *pushout* in a suitable category.

Comparing the three graphs in the rule, one can see that, in order to change the value of the attribute atLoc of the Passenger, the whole hyperedge is deleted and created again: one cannot delete a single attribute, as the resulting structure would not be a legal hypergraph.[2] Instead, the node representing the identity of the passenger is preserved by the rule. Also, all the other items present in

[2] This is a design choice which forbids the simultaneous application of another rule accessing the Passenger. Conceptually, this is equivalent to putting a “lock” on the object whose attribute is changed.

the left-hand side (needed to enforce the preconditions for the application of the rule) are not changed by the rule.

In most cases, it is possible to use a much more concise representation of a rule of this kind, by depicting it as a single graph (the union of L and R), and annotating which items are removed and which are created by the rule. Figure 4 (a) shows an alternative but equivalent graphical representation of the rule of Figure 3 as a degenerate kind of *collaboration diagram* (without sequence numbers, guard conditions, etc.) according to [6].

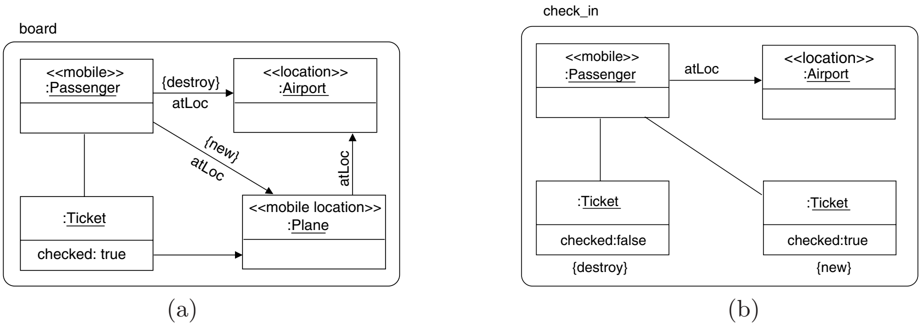


Fig. 4. The rules for boarding (a) and for checking in (b) as collaboration diagrams

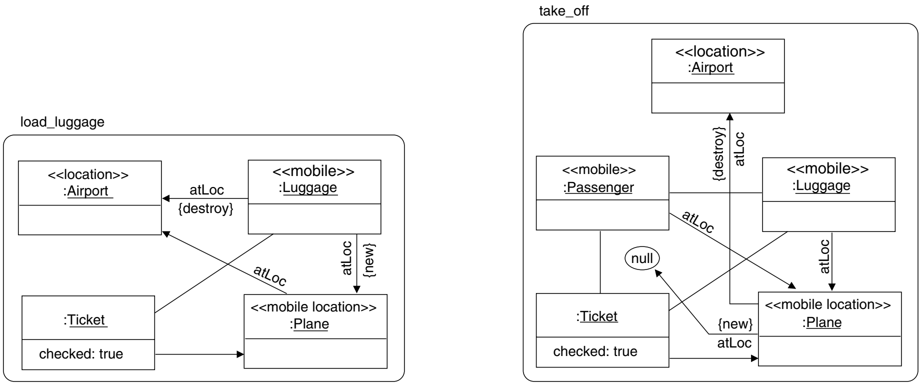
Here the state of the system is represented as an instance diagram, and the items which are deleted by the rule (resp. created) are marked by {destroy} (resp. {new}: beware that these constraints refer to the whole Passenger instance, and not only to the atLoc tentacle). For graph transformation rules with injective right-hand side (and no shared variable, like all those considered here), this representation is equivalent to the one with explicit left-hand side, interface and right-hand side graph, and for the sake of simplicity we will stick to it.

Figure 4 (b) and Figures 5 (a, b) show the rules implementing the remaining three activities of Figure 2, namely check_in, load_luggage and take_off: the corresponding graphical representation can be recovered easily. Notice that the effect of the take_off rule is to change the value of the atLoc attribute of the plane: we set it to null, indicating that the location is not meaningful after taking off; as a different choice we could have used a generic location like Air or Universe.

The next statement, by exploiting definitions and results from the theory of graph transformation, describes the causal relationships among the potential rule applications to the instance graph of Figure 1 (b) (as depicted in Figure 6), showing that the dependencies among activities stated in the diagram of Figure 2 are correctly realized by the proposed implementation.

Proposition 1 (Causal Dependencies Among Rules Implementing Activities). *Given the start instance graph G_0 of Figure 1 (b) and the four graph transformation rules of Figures 4 and 5,*

- *the only rule applicable to G_0 is* check_in, *producing, say, the instance graph G_1;*



(a) (b)

Fig. 5. The rules for loading the luggage (a) and for taking off (b)

$$\begin{tikzcd}
 & & G_1' \arrow[dr, "\text{board}"] & & \\
G_0 \arrow[r, "\text{check\_in}"'] & G_1 \arrow[ur, "\text{load\_luggage}"] \arrow[dr, "\text{board}"'] & & G_2 \arrow[r, "\text{take\_off}"'] & G_3 \\
 & & G_1'' \arrow[ur, "\text{load\_luggage}"'] & &
\end{tikzcd}$$

Fig. 6. Dependencies among the graph transformation rules of the Departure Use Case

- *both* board *and* load_luggage *can be applied to graph* G_1*, in any order or even in parallel, resulting in all cases in the same graph (up to isomorphism), say* G_2*;*
- *rule* take_off *can be applied to* G_2*, but not to any other instance graph generated by the above mentioned rules.*

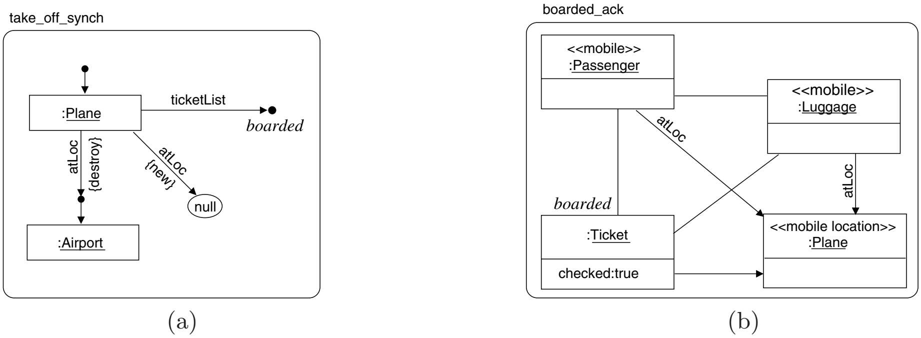
2.3 Enriching the Model with Synchronized Hypergraph Rewriting

Quite obviously, the rule take_off fits in the unrealistic assumption that the flight has only one passenger. Let us discuss how this assumption can be dropped by modeling the fact that the plane takes off only when *all* its passengers and *all* their luggages are boarded.

We shall exploit the expressive power of Synchronized Hypergraph Rewriting [15], an extension of hypergraph rewriting, to model this situation in a very concise way. Intuitively, the plane has as attribute the collection of *all* the tickets for its flight, and when taking off it broadcasts a synchronization request to all the tickets in the collection. Each ticket can synchronize only if its passenger and its luggage are on the plane. If the synchronization fails, the take_off rule cannot be applied. This activity can be considered as an abstraction of the check performed by the hostess/steward before closing the gate.

Conceptually, a graph transformation rule *with synchronization* is a rule where one or more nodes of the left-hand side may be annotated with an *action*.

If the node is a variable, the action is interpreted as a synchronization request issued to the instance which will be bound to the variable when applying the rule. If the annotated node is the source of an instance, the action is interpreted as an acknowledgment issued by that instance. Given an instance graph, a bunch of such rules with synchronization can be applied simultaneously to it only if, besides satisfying the usual conditions for parallel application, all the synchronization requests are properly matched by a corresponding acknowledgment.



(a) (b)

Fig. 7. The rules for taking off while checking that all passengers are on board (a), and for acknowledging the synchronization (b)

To use this mechanism in our case study, let us first assume that at the class diagram level we inserted an association Plane $\overset{1\quad *}{\Longleftrightarrow}$ Ticket with the obvious meaning: we call TicketList the corresponding attribute of a plane. Figure 7 (a) shows rule take_off_synch: the plane takes off, changing its location from the airport to null, only if its request for a synchronization with a *boarded* action is acknowledged by its collection of tickets. In this rule we depict the state as an instance graph, showing explicitly that a node representing the value of the attribute ticketList of the plane is annotated by the *boarded* action. On the other side, according to rule boarded_ack of Figure 7 (b), a ticket can acknowledge a *boarded* action only if its passenger and its luggage are both located on its plane. Here the state is depicted again as an instance *diagram*, and the *boarded* action is manifested on the node representing the identity of the ticket.

To complete the description of the system, we must explain how the tickets for the flight of concern are linked to the ticketList attribute of the plane. In order to obtain the desired synchronization between the plane and all its tickets, we need to assume that there is a subgraph which has, say, one "input node"(the ticketList attribute of the plane) and n "output nodes" (the tickets); furthermore, this subgraph should be able to "match" synchronization requests on its input to corresponding synchronization acknowledgments on its ouputs.

More concretely, this is easily obtained, for example, by assuming that the collection of tickets is a linked list, and by providing rules for propagating the synchronization along the list: this is shown in Figure 8, where the rules should be intended to be parametric with respect to the action *act*.

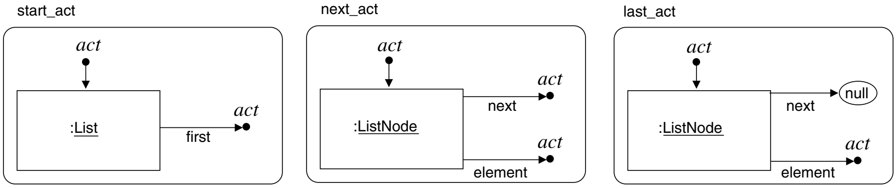


Fig. 8. The rules for broadcasting synchronizations along a linked list

3 A Logic for Graph Transformation Systems

This section presents a slight variation of the behavioral logic for graph transformation systems proposed in [5], adapted to deal with hypergraphs. It is essentially a variant of the propositional μ-calculus (i.e., a temporal logic enriched with fixed-point operators) where propositional symbols range over arbitrary *state predicates*, characterizing static graph properties, which are expressed in a monadic second-order logic.

3.1 A Monadic Second-Order Logic for Graphs

We first introduce the monadic second-order logic $\mathcal{L}2$ for specifying graph properties, i.e.,"static" properties of system states. Quantification is allowed over edges, but not over nodes (as, e.g., in [8]).

Definition 1 (Graph Formulae). *Let $\mathcal{X}_1 = \{x, y, w, \ldots\}$ be a set of (first-order) edge variables and $\mathcal{X}_2 = \{X, Y, W, \ldots\}$ be a set of (second-order) variables ranging over edge sets. The set of* graph formulae *of logic $\mathcal{L}2$ is defined as*

$$\begin{array}{rcll} F & ::= & x = y \;\mid\; x.attrx = y.attry \;\mid & \\ & & type(x) = \ell \;\mid\; x \in X \;\mid & \textit{(Predicates)} \\ & & F \vee F \;\mid\; F \wedge F \;\mid\; F \Rightarrow F \;\mid\; \neg F \;\mid & \textit{(Connectives)} \\ & & \forall x.F \;\mid\; \exists x.F \;\mid\; \forall X.F \;\mid\; \exists X.F & \textit{(Quantifiers)} \end{array}$$

where ℓ belongs to a set Λ of labels, *and attrx, attry to a fixed set of* attribute names*. We denote by free(F) and Free(F) the sets of first-order and second-order variables, respectively, occurring free in F.*

Let G be an instance graph, let F be a graph formula in $\mathcal{L}2$, and let $\sigma : \mathit{free}(F) \rightarrow \mathit{Edges}(G)$ and $\Sigma : \mathit{Free}(F) \rightarrow \mathcal{P}(\mathit{Edges}(G))$ be valuations for the free first- and second-order variables of F, respectively. The *satisfaction relation* $G \models_{\sigma,\Sigma} F$ is defined inductively, in the usual way; for instance

$$\begin{aligned}
G \models_{\sigma,\Sigma} x = y &\iff \text{$\sigma(x)$ and $\sigma(y)$ are the same edge;}\\
G \models_{\sigma,\Sigma} x.attrx = y.attry &\iff \text{edges $\sigma(x)$ and $\sigma(y)$ have attributes (tentacles) $attrx$ and $attry$, respectively, and they point to the same node;}\\
G \models_{\sigma,\Sigma} type(x) = \ell &\iff \text{the object represented by edge $\sigma(x)$ is an instance of class ℓ;}\\
G \models_{\sigma,\Sigma} x \in X &\iff \text{edge $\sigma(x)$ belongs to the set of edges $\Sigma(X)$.}
\end{aligned}$$

If the formula F is closed then we will write $G \models F$ instead of $G \models_{\emptyset,\emptyset} F$. As an example, the formula $\exists p.\exists t.type(p) = Passenger \wedge type(t) = Ticket \wedge p.ticket = t.self$ holds true in the instance graph of Figure 1, using the assumption that the only source tentacle of each hyperedge is implicitly labeled by self.

We shall freely use the following obvious abbreviations for graph formulae

$$\begin{aligned}
\forall x : T \,.\, \phi &\triangleq \forall x \,.\, type(x) = T \Rightarrow \phi\\
\exists x : T \,.\, \phi &\triangleq \exists x \,.\, type(x) = T \wedge \phi\\
x.attr = y &\triangleq x.attr = y.self
\end{aligned}$$

and, in any context where a graph formula is expected,

$$\begin{aligned}
x.attr &\triangleq x.attr = true.self\\
\neg(x.attr) &\triangleq x.attr = false.self
\end{aligned}$$

where the constants *true* and *false* are interpreted over the (unary) hyperedges encoding the booleans, which we assume to be included in every instance graph.

3.2 Introducing a Temporal Dimension

The behavioral logic for GTSs, called $\mu\mathcal{L}2$, is a variant of the propositional μ-calculus where propositional symbols range over formulae from $\mathcal{L}2$.

Definition 2 (Logic Over GTSs). *The syntax of $\mu\mathcal{L}2$ formulae is given by*

$$f ::= \ A \mid Z \mid \Diamond f \mid \Box f \mid \neg f \mid f_1 \vee f_2 \mid f_1 \wedge f_2 \mid \mu Z.f \mid \nu Z.f$$

where A ranges over closed formulae in $\mathcal{L}2$ and $Z \in \mathcal{Z}$ are proposition variables.

The formulae are evaluated over a *graph transition system* $T = (Q, \rightarrow)$, i.e., a transition system where the set of states Q consists of (isomorphism classes of) graphs. This can be thought of as the abstract representation of the behavior of a graph grammar $\mathcal{G}$: states in Q are (isomorphism classes) of graphs reachable in $\mathcal{G}$ and two states q_1 and q_2 are related, i.e., $q_1 \rightarrow q_2$, if q_2 is reachable from q_1 via a rewriting step in $\mathcal{G}$.

Intuitively, an atomic proposition A holds in a state q if $q \models A$ according to the satisfaction relation of the previous section. A formula $\Diamond f$ / $\Box f$ holds in a state q if some / any single step leads to a state where f holds. Note that (as in [19]) the operators $\Box$ and $\Diamond$ only refer to the next step and not (as defined elsewhere) to the whole computation. The connectives $\neg, \vee, \wedge$ are interpreted in the usual way. The formulae $\mu Z.f$ and $\nu Z.f$ represent the *least* and *greatest fixed point* over Z, respectively. When a transition system T has a distinguished *initial state* q_0, we say that T *satisfies a (closed) formula* f, written $T \models f$, if the initial state q_0 of T satisfies f. Since the logic is classical, $\Diamond$ and ν could be defined in terms of $\Box$ and μ.

Since properties of the form "eventually ϕ", i.e., $\mu Z.(\phi \vee \Diamond Z)$, and "always ϕ", i.e., $\nu Z.(\phi \wedge \Box Z)$, will be often used, we introduce the abbreviations

$$\begin{aligned} \Diamond^* \phi &\triangleq \mu Z\,.\,(\phi \vee \Diamond Z) \\ \Box^* \phi &\triangleq \nu Z\,.\,(\phi \wedge \Box Z) \end{aligned}$$

3.3 Specifying Some Properties of the Airport Scenario

In this section we discuss how some properties concerned with the Airport Case Study can be modeled in our logic. As mentioned before, the main limitation of the logic $\mu\mathcal{L}2$ resides in its propositional nature which prevents from describing the evolution of an object in time. We briefly discuss how this limitation can be overcome by considering a non-propositional extension of the temporal logic.

Using the Logic $\mu\mathcal{L}2$. The logic $\mu\mathcal{L}2$ can be used to express properties about the structure of system state, possibly at different instants.

- The plane leaves only if all passengers are aboard
  Fixed an edge pl : *Plane* representing a plane, the formula

  $$\begin{aligned} \phi(pl) \quad \triangleq \quad & \exists p\colon \mathsf{Passenger}\,.\,(\exists tk\colon \mathsf{Ticket}\,.\,(p.\mathsf{ticket} = tk \wedge tk.\mathsf{plane} = \\ & pl \wedge tk.\mathsf{checked} \wedge p.\mathsf{atLoc} \neq pl)) \end{aligned}$$

  means that there is a passenger p having a ticket tk associated with the plane pl, the ticket is checked but the passenger is not aboard. Hence the desired property can be expressed by saying that this can never happen for any plane which is on air, i.e., such that its atLoc attribute is *null*

  $$\Box^*(\forall pl\colon \mathsf{Plane}\,.\,(pl.\mathsf{atLoc} = \mathit{null} \Rightarrow \neg\phi(pl)))$$

- A passenger can eat only on air
  This property is expressed by the formula:

  $$\Box^*(\forall p\colon \mathsf{Passenger}\,.\,\forall pl\colon \mathsf{Plane}\,.\,((p.\mathsf{eat} \wedge p.\mathsf{atLoc} = pl) \Rightarrow pl.\mathsf{atLoc} = \mathit{null}))$$

  which says, assuming the existence of a tentacle eat, that it is always true that if a passenger is eating on a plane, then the plane is flying.

The validity of the above formulae over a finite-state system, like the Airport Case Study with a given initial state, can be checked by using a technique based on the unfolding semantics of GTSs and inspired to the approach originally developed by McMillan for Petri nets [20]. Recall that the unfolding construction for GTSs produces a structure which fully describes the concurrent behavior of the system, including all possible steps and their mutual dependencies, as well as all reachable states. However, the unfolding is infinite for non-trivial systems, and cannot be used directly for model-checking purposes. An algorithm proposed in [4] allows for the construction of a finite initial fragment of the unfolding of the given system which is *complete*, i.e., which provides full information about the system as far as reachability (and other) properties are concerned. Once it has been constructed, the prefix can be used to verify properties of the reachable states, expressed in the logic $\mathcal{L}2$. This is done by exploiting both the graphical structure underlying the prefix and the concurrency information it provides.

We mention that approximated techniques, also based on the unfolding semantics of GTSs, are available for systems which are not finite-state. In this case, finite under- and over-approximations of the unfolding can be constructed, which are used to check properties of a graph transformation system, like safety and liveness properties, expressed in suitable fragments of $\mu\mathcal{L}2$ [5].

A More General Logic. By experimenting with the Airport Case Study, it turns out that some interesting properties of the system cannot be expressed in $\mu\mathcal{L}2$ essentially because of its propositional nature. Take, for instance, the property "*All boarded passengers arrive at destination*". The corresponding formula should say that it is always true that, given any passenger, if *in a certain state* the passenger is boarded then *later, eventually* the passenger will arrive at its destination. This formula would have the shape

$$\Box^*(\forall p\colon \mathsf{Passenger}\,.\, p\ \mathit{is\ boarded} \Rightarrow \Diamond^*(p\ \mathit{at\ destination}))$$

which is not expressible in $\mu\mathcal{L}2$ due to the presence of the modal operator "$\Diamond^*$" in the scope of the quantifier "$\forall$".

The problem can be overcome by considering a more general, non-propositional temporal graph logic, where quantifiers and temporal operators can be interleaved. A possible syntax is given below, where the operators $\Box^*$ and $\Diamond^*$ are taken as primitive.

$$\begin{array}{rll} F ::= & x = y \mid x.attrx = y.attry \mid & \\ & \mathit{type}(x) = \ell \mid x \in X \mid & \text{(Predicates)} \\ & F \vee F \mid F \wedge F \mid F \Rightarrow F \mid \neg F \mid & \text{(Connectives)} \\ & \forall x.F \mid \exists x.F \mid \forall X.F \mid \exists X.F & \text{(Quantifiers)} \\ & \Diamond^* F \mid \Box^* F & \text{(Temporal Operators)} \end{array}$$

As before *attrx*, *attry* belong to a fixed set of *attribute names*, and x, X are first- and second-order variables, ranging over edges and set of edges, respectively.

The semantics of such logic can be defined by mimicking what is done, e.g., for first-order or second-order modal and temporal logics (see [16], or the more recent [9], where a graph logic is considered). Roughly, the logic is interpreted over a Kripke structure or a transition system where states are (first- or second-order) models. Since the logic allows to track the evolution of an individual, when a state q_1 can evolve to q_2, there must exist an explicit relationship among the elements of the models underlying such states.

More precisely, our logic could be interpreted over an *extended graph transition system* (Q, F), with Q a set of graphs and F a set of triples (q_1, f, q_2), where q_1, q_2 are states in Q and $f : q_1 \to q_2$ is a partial graph morphism. The presence of a triple (q_1, f, q_2) intuitively means that the graph q_1 can evolve to q_2. The function f relates any item in the graph q_1 which is *not* deleted by the rewriting step to the corresponding item in q_2.

By using this extended logic, several properties previously not expressible in $\mu\mathcal{L}2$ can now be easily modeled.

- All boarded passengers arrive at destination
  This property can be encoded by the following formula

$$\Box^*(\forall p\colon \mathsf{Passenger}\,.\,\forall pl\colon \mathsf{Plane}\,.\,((p.\mathsf{atLoc} = pl) \Rightarrow \exists a\colon \mathsf{Airport}\,.\,(pl.\mathsf{dest} = a \land \Diamond^*(p.\mathsf{atLoc} = a))))$$

  which says that, in any state, if a passenger p is boarded on a plane pl, whose destination is airport a, then the passenger p will eventually arrive at a.
- All passengers go on board
  This property can be encoded by the following formula

$$\Box^*(\forall p\colon \mathsf{Passenger}\,.\,\forall t\colon \mathsf{Ticket}\,.\,\forall pl\colon \mathsf{Plane}\,.\,(p.\mathsf{ticket} = t \land t.\mathsf{plane} = pl \Rightarrow \Diamond^*(p.\mathsf{atLoc} = pl)))$$

  which says that, in any state, each passenger with a ticket associated to a plane pl will eventually board on pl.
- Airports cannot move
  This property can be encoded by the following formula

$$\Box^*(\forall a\colon \mathsf{Airport}\,.\,\forall x\,.\,(a.\mathsf{atLoc} = x \Rightarrow \Box^*(a.\mathsf{atLoc} = x)))$$

  which says that an airport which is at some location will always stay there.
- Passengers change airport only by plane
  This property can be encoded by the following formula

$$\Box^*(\forall p\colon \mathsf{Passenger}\,.\,\forall a_1\colon \mathsf{Airport}\,.\,(p.\mathsf{atLoc} = a_1 \land \Box^*(\neg\exists pl\colon \mathsf{Plane}\,.\,(p.\mathsf{atLoc} = pl)) \Rightarrow \Box^*(\forall a_2\colon \mathsf{Airport}\,.\,((p.\mathsf{atLoc} = a_2) \Rightarrow a_2 = a_1))))$$

  which says that a passenger which is at an airport a_1 and which does not take any plane will be always in a_1.
- Baggage travels with passengers (each bag with its owner)
  This property can be encoded by the following formula

$$\Box^*(\forall p\colon \mathsf{Passenger}\,.\,\forall l\colon \mathsf{Luggage}\,.\,\forall pl\colon \mathsf{Plane}\,.\,(p.\mathsf{atLoc} = pl \land p.\mathsf{luggage} = l \Rightarrow \Diamond^*(p.\mathsf{atLoc} = pl \land l.\mathsf{atLoc} = pl))$$

which says that, in any state, if a passenger p has a luggage l and it boards on a plane pl then, eventually, also the luggage will be in pl together with the passenger.

- Passengers change location with their plane
  This property is interpreted as "a passenger on a plane will reach the same destination as the plane itself". This can be encoded by the following formula

$$\Box^*(\forall p\colon \mathsf{Passenger}\,.\,\forall pl\colon \mathsf{Plane}\,.\,\forall a\colon \mathsf{Airport}\,.\,(p.\mathsf{atLoc} = pl \ \wedge\ pl.\mathsf{dest} = a \Rightarrow \Diamond^*(p.\mathsf{atLoc} = a)))$$

  which says that, in any state, if a passenger p is on a plane pl with destination airport a then p eventually reaches airport a.

Unfortunately, the unfolding-based verification techniques mentioned for $\mu\mathcal{L}2$ do not immediately extend to this more general framework. Understanding to what extent such techniques can be adapted to the new framework is an open and stimulating direction of further research.

4 Conclusions

We presented a general approach to the specification and verification of systems modeled using UML diagrams, interpreted as graph transformation systems. More precisely, we discussed how to interpret instance diagrams as graphs, and how to implement the activities of an activity diagram as graph transformation rules. A temporal logic over monadic second-order graph predicates allows for formalizing relevant properties of the resulting system, and for fragments of such logic automatic verification techniques are available. The overall approach was presented quite informally, using a simple case study as running example.

The theoretical foundations of the methodology for verifying graph transformation systems presented in the second part of the paper are already quite well developed [3, 4, 5]. The most valuable contribution of this paper, in our view, is the idea of applying it for the verification of systems synthesized from a UML specification. This allowed us to identify some weaknesses of the approach, as described in the previous section, that we intend to address in the next future by generalizing the verification approach to more expressive logics and to graph transformation systems with synchronization.

Concerning the modeling of UML diagrams as graph transformation systems, the informal approach we discussed is clearly very preliminary, as it addresses only (restricted forms of) two kinds of diagrams. Nevertheless, we are confident, also on the basis of other contributions concerned with this topic [10, 11, 12, 13, 17, 18], that such an approach can be extended to cover a meaningful part of the UML. This represents another interesting topic for future research.

References

1. The AGILE project home page, `http://siskin.pst.informatik.uni-muenchen.de/projekte/agile/`, 2004.

2. L. Andrade, P. Baldan, H. Baumeister, et al. AGILE: Software architecture for mobility. In M. Wirsing, D. Pattinson, and R. Hennicker, editors, *Proceedings of the 16th International Workshop on Recent Trends in Algebraic Develeopment Techniques (WADT 2002)*, volume 2755 of *LNCS*, pages 1–33. Springer, 2003.
3. P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In K.G. Larsen and M. Nielsen, editors, *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR 2001)*, volume 2154 of *LNCS*, pages 381–395. Springer, 2001.
4. P. Baldan, A. Corradini, and B. König. Verifying finite-state graph grammars: an unfolding-based approach. In P. Gardner and N. Yoshida, editors, *Proceedings of the 15th International Conference on Concurrency Theory (CONCUR 2004)*, LNCS. Springer, 2004.
5. P. Baldan and B. König. Approximating the behaviour of graph transformation systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the First International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *LNCS*, pages 14–30. Springer, 2002.
6. A. Corradini, R. Heckel, and U. Montanari. Graphical operational semantics. In A. Corradini and R. Heckel, editors, *Proceedings of the ICALP Workshop on Graph Transformations and Visual Modeling Techniques (GT-VMT 2000)*, volume 8 of *Proceedings in Informatics*, pages 411–418. Carleton Scientific, 2000.
7. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*. World Scientific, 1997.
8. B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*. World Scientific, 1997.
9. D. Distefano, A. Rensink, and J.-P. Katoen. Model checking dynamic allocation and deallocation. CTIT Technical Report TR–CTIT–01–40, Department of Computer Science, University of Twente, 2002.
10. G. Engels, J.H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioural diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *Proceedings of the Third International Conference on the Unified Modeling Language (UML 2000)*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.
11. G.L. Ferrari, U. Montanari, and E. Tuosto. Graph-based models of internetworking systems. In B.K. Aichernig and T. Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support*, volume 2757 of *LNCS*, pages 242–266. Springer, 2003.
12. M. Gogolla. Graph transformations on the UML metamodel. In A. Corradini and R. Heckel, editors, *Proceedings of the ICALP Workshop on Graph Transformations and Visual Modeling Techniques (GT-VMT 2000)*, volume 8 of *Proceedings in Informatics*, pages 359–371. Carleton Scientific, 2000.
13. M. Gogolla, P. Ziemann, and S. Kuske. Towards an integrated graph based semantics for UML. In P. Bottoni and M. Minas, editors, *Proceedings of the ICGT Workshop on Graph Transformations and Visual Modeling Techniques (GT-VMT 2002)*, volume 72 of *ENTCS*. Elsevier, 2003.

14. R. Heckel, J.M. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In A. Corradini, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proceedings of the First International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *LNCS*, pages 161–176. Springer, 2002.
15. D. Hirsch and U. Montanari. Synchronized hyperedge replacement with name mobility. In K.G. Larsen and M. Nielsen, editors, *Proceedings of the 12th International Conference on Concurrency Theory (CONCUR 2001)*, volume 2154 of *LNCS*, pages 121–136. Springer, 2001.
16. G.E. Hughes and M.J. Cresswell. *A new introduction to modal logic.* Routledge, 1996.
17. S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In A. Haeberer, editor, *Proceedings of the Fourth International Conference on the Unified Modeling Language (UML 2001)*, volume 2185 of *LNCS*, pages 241–256. Springer, 2001.
18. S. Kuske, M. Gogolla, R. Kollmann, and H.J. Kreowski. An integrated semantics for UML class, object and state diagrams based on graph transformation. In *Proceedings of the Third International Conference on Integrated Formal Methods (IFM 2002)*, volume 2335 of *LNCS*, pages 11–28. Springer, 2002.
19. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, 1995.
20. K.L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.
21. D. Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 2: Applications, Languages, and Tools*. World Scientific, 1999.
22. J. Rumbaugh, I. Jacobson, and G. Book. *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

Mobile UML Statecharts with Localities*

Diego Latella[1], Mieke Massink[1], Hubert Baumeister[2], and Martin Wirsing[2]

[1] CNR, Istituto di Scienza e Tecnologie dell'Informazione,
Via Moruzzi 1, I56124 Pisa, Italy
{Diego.Latella, Mieke.Massink}@isti.cnr.it
[2] LMU, Institut für Informatik, Oettingenstr. 67, D-80538 München, Germany
{Hubert.Baumeister, Martin.Wirsing}@ifi.lmu.de

Abstract. In this paper an extension of a behavioural subset of UML statecharts for mobile computations is proposed. We study collections of UML objects whose behaviour is given by statecharts. Each object resides in a given place, and a collection of such places forms a network. Objects are aware of the *localities* of other objects, i.e. the logical names of the places where the latter reside, but not of the physical name of such places. In addition to their usual capabilities, such as sending messages etc., objects can move between places and create and destroy places, which may result in a deep reconfiguration of the network. A formal semantics is presented for this mobility extension which builds upon a core semantics definition of statecharts without mobility which we have used successfully in several contexts in the past years. An example of a model of a network service which exploits mobility for resource usage balance is provided using the proposed extension of UML statecharts.

1 Introduction

Mobility plays a major role in the programming of nowadays network-based services. The Unified Modelling Language (UML) is the de facto standard graphical modelling language for object-oriented software and systems [17]. It has been specifically designed for visualising, specifying, constructing, and documenting several aspects of—or views on—systems. In this paper we focus on a behavioural subset of UML statecharts (UMLSCs) and in particular on a powerful extension of this notation in order to deal with a notion of mobility which is sometimes referred to as *mobile computation* and requires computing elements to be able to migrate from one node to another within a network [4], as opposed to *mobile computing*, where the focus is instead on dynamic communication structures. We address mobile computing in the context of UMLSCs in a companion paper [12].

In this paper we assume that a system is modelled as a *dynamic* collection of (cooperating, autonomous) *objects*. In order to express mobile computations we assume that each object is located at exactly one network node, or *place* as

* This work has been carried out in the context of Project EU-IST IST-2001-32747 Architectures for Mobility (AGILE).

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 34–58, 2004.

we shall call it in the sequel. A *network* is a collection of places. Note that for simplicity a network has a flat hierarchy similar to the approach used in KLAIM [5], that is, places cannot contain places.

The attributes of an object may contain values of basic datatypes, like integer, boolean e.t.c., references to other objects, and references to network nodes. The behaviour of an object is given by a UMLSC. Objects can move among places and objects and places can be created or destroyed.

Objects are not aware of the physical names of places; they make reference only to logical names, also called *localities*, which play the same role as symbolic addresses in the Internet. Consequently, each place is also equipped with an *allocation environment* which maps localities to the physical names of places. The architecture of the network is dynamic and is implicitly defined by the information encapsulated in the allocation environments of all the places belonging to the network; such information collectively defines the set of places each place is "in touch with". The choice of objects being unaware of place physical names has been inspired by the work on KLAIM [5].

Example. Consider a simplified model of a resource based compute server. Depending on the resources needed for a computation, the computation should be performed at different network places or being split among different network places. Figure 1 shows the class diagram using the stereotype ≪mobile≫ introduced in [2] to indicate classes whose objects can move between places.

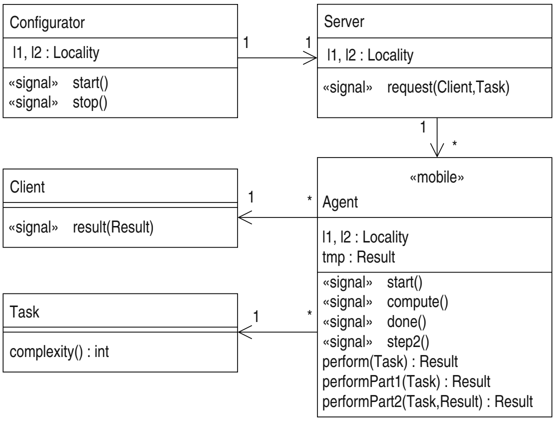


Fig. 1. Class diagram for a simple compute server

The behaviour of the configurator, server, and agents is given by the statecharts in Figs. 2, 3, and 4. The configurator first creates two network places where the computation should take place (cf. Fig. 2). During the creation of the network places also two new localities are created that refer to these places and which are stored in attributes l_1 and l_2. The configurator then exports the place where the configurator is located (given by variable atLoc) to the newly created places, i.e. the locality home at the new places refers to the place of the configurator. Next, the server is created and initialised with the two localities l_1 and l_2.

When the server receives a request by a client to perform a task, an agent is created to fulfil the request (cf. Fig. 3). The agent moves to the different network places depending on the complexity of the computation task (cf. Fig. 4). In case of a simple task, the agent stays at the place of the server, computes the result of the task, and sends the result back to client. In case of a more complex task, the agent first moves to the place referred to by l_1, performs the computation there, and then sends the result back to the client who stayed at the place referred to by locality home. In the most complex case, the agent first performs part of the computation at the place referred to by locality l_1 and then the second part of the computation at locality l_2 before sending the result to the client at locality home. After the agent has done its work he destroys itself.

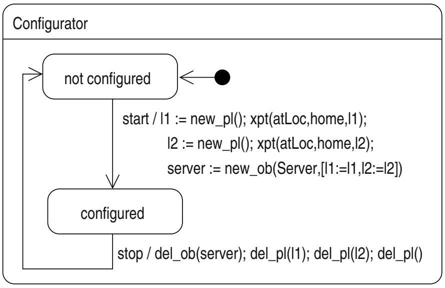

Configurator
not configured
start / l1 := new_pl(); xpt(atLoc,home,l1);
l2 := new_pl(); xpt(atLoc,home,l2);
server := new_ob(Server,[l1:=l1,l2:=l2])
configured
stop / del_ob(server); del_pl(l1); del_pl(l2); del_pl()

Fig. 2. Statechart for class Configurator of Fig. 1

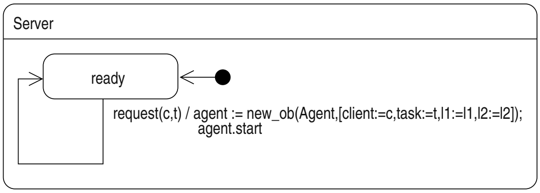

Server
ready
request(c,t) / agent := new_ob(Agent,[client:=c,task:=t,l1:=l1,l2:=l2]);
agent.start

Fig. 3. Statechart for class Server of Fig. 1

In the present paper, we propose a formal operational semantics for the mobility extension of UMLSCs briefly discussed above. The operational semantics is built upon previous work of Latella et al. on formalising the semantics of UMLSCs [11, 6], which in turn was inspired by the work of Mikk [16] on Harel statecharts.

Several other proposals for formal semantics of UMLSCs can be found in the literature, e.g. [19, 3, 14, 13, 18]. None of these approaches deals with mobility; we refer to [6] for a comparison with our previous work. An approach similar to ours is [10] in which ambients [4] are added to Interacting State Machines

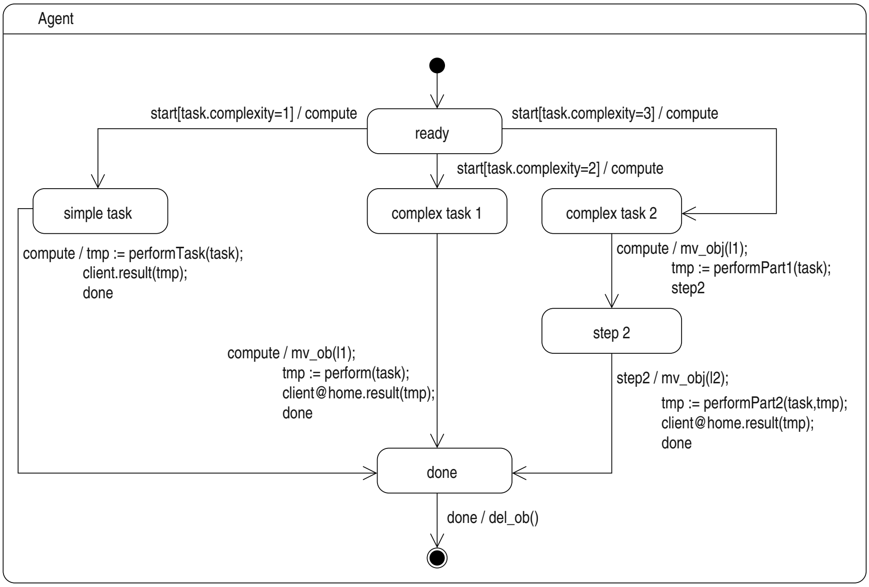


Fig. 4. Statechart for class Agent of Fig. 1

(ISMs). The differences are that this approach is not intended to model UMLSCs and therefore does not deal directly with features of UMLSCs, like composite states and concurrent substates, and that ISMs have a hierarchical structure of places while our place structure is flat. Another approach is [8] where UML state machines with mobility are translated to MTLA formulas [15] to study refinement of state machines.

Several proposals for extending the UML with mobility and/or agent notions are present in the literature, e.g. [2, 9, 7, 1]. In [2] a proposal for extending UML activity diagrams with mobility notions is presented while in [9] mobility in UML sequence diagrams is addressed. In [7] an extension of UML in order to describe agent interaction is proposed which does not address mobility explicitly. Mobile agents in UML are the focus of [1]. All the above mentioned proposals deal with notational extensions of UML performed mainly by means of UML extension mechanisms (stereotypes, tagged-values, etc.) and do not address issues of formal semantics. We are not aware of any proposal for a formal semantics of objects, object management, mobility and network configuration in the context of UML statecharts.

The paper is organised as follows: Section 2 briefly addresses the general framework of our approach, including our modelling assumptions and an informal introduction of the basic notions of hierarchical automata for UMLSCs. Section 3 describes the kind of actions which can label UMLSCs transitions and the notion of network specification. The formal semantics is given in Sect. 4 and conclusions are drawn in Sect. 5. Finally, for the interested reader, the appendix contains the formal semantics of hierarchical automata.

2 Basic Framework

In this section we set the basic framework of our work. In particular we present a brief description of the modelling assumptions on which we base our approach and an informal introduction to the basic notions of the computational framework of hierarchical automata, which we use as an abstract syntax for the definition of UMLSCs semantics. All technical details on hierarchical automata for UMLSCs can be found in [11, 6] and in the appendix.

2.1 Object Model

In this paper we assume that a system is modelled as a *dynamic* collection of (cooperating, autonomous) *objects* and that the behaviour of each object is specified by a UMLSC (more than one single object may have its behaviour specified by the same UMLSC). More precisely, we assume that a networked system is specified by a *static* collection $\{SC_1, \ldots, SC_c\}$ of UMLSCs. In order to express mobile computations, we explicitly model network nodes, or *places* as we shall call them in the sequel, and we define a *network* as a collection of places. The set of places of a network may dynamically change during the evolution of the network.

Each place has a set of objects residing therein and is uniquely identified within the network by its *physical name*. A physical name can be thought of as an IP address in the context of the Internet. The objects residing within a place are uniquely identified by their *object names* within that place. The set of objects residing in a given place may dynamically change during the evolution of the network.

Objects cannot directly refer to the places' physical names; instead, they use logical names, called *localities*, which are mapped to physical names by the *allocation environment* of the place where the object resides. In addition, each object may have a private allocation environment. The network structure is not given explicitly, but implicitly by the information encapsulated in the allocation environments, which defines the set of places a place is "in touch with".

The above assumptions are quite realistic and rather common in networked systems, in particular unawareness of physical names of places. In this choice we have been inspired by the work on KLAIM [5] of which our model also shares the flat structure of places. Other proposals, like for instance [4], assume a hierarchical structure. We choose a flat structure for the sake of simplicity.

2.2 Hierarchical Automata

As briefly mentioned in Sect. 2.1, in our approach, a networked system is specified by a finite collection of UMLSCs $\{SC_1, \ldots, SC_c\}$. We use hierarchical automata (HAs) [16] as an abstract syntax for UMLSCs. HAs for UMLSCs have been introduced in previous work of co-authors of the present paper ([11, 6]). In this section we recall, informally, only the main notions which are necessary for the understanding of the paper. The reader interested in the detailed formal definitions concerning UMLSCs, like the definition of the behavioural semantics, is referred to the appendix. Such definition is essentially an orthogonal extension of the original formalisation of UMLSCs semantics of [11, 6], where mobility was *not* addressed.

Informally, a HA is composed of a collection of simple sequential automata related by a *refinement function* which imposes on the HA the hierarchical state nesting-structure of the associated statechart. Inter-level transitions are encoded by means of proper annotations in transition labels.

The operational semantics of HAs, which we shall define later on in the present paper, makes use of what we call the *Core Semantics* of HAs. The Core Semantics of a HA H characterises the relation $H :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_L \mathcal{C}'$ with the following intended meaning: whenever the current *state configuration* of H is $\mathcal{C}$, its current variables/values binding is *store* β, and *event* ev is fed to H's *state-machine*, the *transitions* in L may fire bringing H to configuration $\mathcal{C}'$; Ac is the sequence of *actions* to be executed—actually an interleaving of the action sequences labelling the transitions in L—and ξ records the binding of the parameters occurring in the *triggers* of such transitions with the corresponding values in ev.

Thus the role of the Core Semantics is the characterisation of the set of transitions to be fired, their related actions, and the resulting configuration. All issues of (action) ordering, concurrency, and non-determinism within single statecharts are dealt with by the Core Semantics. Although essential for the definition of the formal semantics, all the above issues are technically quite orthogonal to mobility and dynamic network/object management.

Another issue which deserves to be briefly addressed here is the way in which we deal with the so called *input-queue* of UMLSCs, i.e. their "external environment". In the standard definition of UML statecharts semantics [17], a *scheduler* is in charge of selecting an event from the input-queue of an object, feeding it into the associated state-machine, and letting such a machine produce a STEP transition. Such a STEP transition corresponds to the firing of a maximal set of enabled non-conflicting transitions of the statechart associated to the object, provided that certain transition priority constraints are not violated. After such transitions are fired and when the execution of all the actions labelling them is completed, the STEP itself is completed and the scheduler can choose another event from a queue and start the next cycle. While in classical statecharts the external environment is modelled by a set, in the definition of UML statecharts, the nature of the input-queue of a statechart is not specified; in particular, the management policy of such a queue is not defined. In our overall approach to UMLSCs semantics definition, we choose *not* to fix any particular semantics, such as set, or multi-set or FIFO-queue etc., but to model the input queue in a policy-independent way, freely using a notion of abstract data types. In the following we assume that for set D, Θ_D denotes the class of all structures of a certain kind (like FIFO queues, or multi-sets, or sets) over D and we assume to have basic operations for manipulating such structures. In particular, in the present paper, we let $\mathsf{Add}\ d\ \mathcal{D}$ denote the structure obtained inserting element d in structure $\mathcal{D}$ and the predicate $(\mathsf{Sel}\ \mathcal{D}\ d\ \mathcal{D}')$ state that $\mathcal{D}'$ is the structure resulting from selecting d from $\mathcal{D}$; of course, the selection policy depends on the choice for the particular semantics. We assume that if $\mathcal{D}$ is the empty structure, denoted by $\langle\rangle$, then $(\mathsf{Sel}\ \mathcal{D}\ d\ \mathcal{D}')$ yields FALSE for all d and $\mathcal{D}'$. We shall often speak of the *input queue*, or simply *queue*, by that meaning a structure in Θ_D, abstracting from the particular choice for the semantics of Θ_D.

We shall refer to the set $\{H_1, \ldots, H_c\}$ of HAs associated to $SC_1, \ldots, SC_c$. Conf_H will denote the set of all state configurations of HA H and we shall assume that for every set $\{H_1, \ldots, H_c\}$ of HAs, there exists a distinguished element $\mathcal{C}_{\mathsf{err}}$ such that $\mathcal{C}_{\mathsf{err}} \notin \bigcup_{j=1}^{c} \mathrm{Conf}_{H_j}$.

3 Action Language and Network Specifications

In this section we introduce the syntax and informal semantics of HAs transition *actions* and *triggers*. Moreover, we formalise the notion of *network specification.*

3.1 Actions

The action side of transition t, i.e. $AC\ t$, is an *action*, and its abstract syntax is shown in Fig. 5. In our extension we will deal with place physical names, place logical names, object names, method names, and with variables[1]; moreover, parameters may occur in method activations. Consequently proper countable, mutually disjoint sets—$\mathcal{Z}_P, \mathcal{Z}_L, \mathcal{Z}_O, \mathcal{Z}_M, \mathsf{Var}$, and Par, respectively—are introduced for them. Moreover we assume that object names form a $\mathcal{Z}_P$-indexed family of disjoint sets.

$$\begin{array}{rl} Ac ::= & var := exp \mid obj@loc_1.meth(exp) \mid \mathsf{mv\_ob}(obj@loc_1, loc_2) \mid \mathsf{mv\_cl}(obj@loc_1, loc_2) \\ \mid & var := \mathsf{new\_ob}(H_0, \mathcal{C}_0, \beta_0, \mathcal{E}_0)@loc_1 \mid var := \mathsf{new\_cl}(H_0, \mathcal{C}_0, \beta_0, \mathcal{E}_0)@loc_1 \\ \mid & \mathsf{del\_ob}(obj@loc_1) \mid var := \mathsf{new\_pl}() \mid \mathsf{del\_pl}(loc) \mid \mathsf{xpt}(loc_1, loc_2, loc_3) \mid Ac; Ac \end{array}$$

where $var \in \mathsf{Var}$, $exp \in \mathsf{Var} \cup \mathsf{Par} \cup \mathcal{Z}_L$, $obj \in \mathsf{Var} \cup \mathsf{Par}$, $loc, loc_i \in \mathsf{Var} \cup \mathsf{Par} \cup \mathcal{Z}_L$ for $i \in \{1, 2, 3\}$, $meth \in \mathcal{Z}_M$, and Var *(variable identifiers)* including self and atLoc, Par *(parameter identifiers)*, $\mathcal{Z}_L$ *(place logical names—i.e. localities)* including here, and $\mathcal{Z}_M$ *(method names)* are countable, mutually disjoint sets.

Fig. 5. Abstract syntax of actions

In the following we informally describe the meaning of the various actions, together with simple static semantics constraints. The formal definition of the action semantics will be addressed later on in the paper while we refrain from giving a formal definition of the static semantics since the static semantics is not very relevant for the purpose of the present paper. Consequently we assume that all variables used in a statechart are declared in the associated object definition and that all actions are type-correct.

[1] We will use "variable" and "attribute" as synonym in this paper.

By $var := exp$, variable var is assigned the value of exp in the current store of the object where the action is executed. Only attribute names of the current object are allowed as variables; i.e. variables can not refer to attributes of different objects. Of course different objects (or even HAs) can use the same variable name which will be bound to possibly different values by different local stores. Reserved, read-only, variables self and atLoc are always bound to the name of the object in the store of that object, and, similarly, the reserved locality here is always bound to the physical place name in the allocation environment of that place.

Action $obj@loc_1.meth(exp)$ sends an *asynchronous message meth* with (optional parameter-)value exp to object obj residing in (the place referred to by) locality loc_1; the following short-hands are provided: $obj.meth(exp)$ is used instead of $obj@\mathsf{atLoc}.meth(exp)$ and $meth(exp)$ instead of $\mathsf{self}.meth(exp)$.

Action $\mathsf{mv\_ob}(obj@loc_1, loc_2)$ makes object obj migrate from locality loc_1 to locality loc_2—notice that the moved object can also be the object executing the action, in which case the short-hand $\mathsf{mv\_ob}(loc_2)$ can be used.

Action $var := \mathsf{new\_ob}(H_0, \mathcal{C}_0, \beta_0, \mathcal{E}_0)@loc_1$ creates a new object the behaviour of which is determined by the HA referred to by H_0. The name of the newly created object will be bound to var in the store of the object executing the action, i.e. the *creator*. The initial configuration $\mathcal{C}_0$ must belong to Conf_{H_0}; for the specification of the initial store β_0 we use the notation $[var_1 := exp_1, \ldots, var_n := exp_n]$, where $exp_1, \ldots exp_n \in \mathsf{Var} \cup \mathsf{Par} \cup \mathcal{Z}_L$, and $var_1, \ldots, var_n \in \mathsf{Var} \setminus \{\mathsf{self}, \mathsf{atLoc}\}$ are attribute names of the created object.[2] The initial configuration, store, and input-queue $\mathcal{C}_0, \beta_0, \mathcal{E}_0$ are optional. If absent, the initial configuration indicated in the definition of the HA referred to by H_0 is used for $\mathcal{C}_0$ while $\mathcal{E}_0$ is empty and β_0 binds only self to the name of the created object and atLoc to the locality where it resides. The newly created object will (initially) reside in locality loc_1. If loc is atLoc or here, the short-hand $var := \mathsf{new\_ob}(H_0, \mathcal{C}_0, \beta_0, \mathcal{E}_0)$ can be used.

The references of an object to localities are normally resolved via the allocation environment of the place where the object resides when the action is executed which uses such references. Moreover, a closure-like version of object creation ($\mathsf{new\_cl}$) and moving ($\mathsf{mv\_cl}$) actions is provided; the allocation environment of the place where the *creator* resides will be inherited by the created object as its private allocation environment; similarly, the allocation environment of the place where the moved object was residing will be inherited by the moved object and extended with its private allocation environment, if any.

By $\mathsf{del\_ob}(obj@loc_1)$ an object kills object obj residing in locality loc_1—notice that the object destroyed can also be the object executing the action, in which case the shorthand $\mathsf{del\_ob}()$ can be used.

[2] Notice that in the specification of β_0 the creator object will access attributes of the created object (i.e. $var_1, \ldots, var_n$); this is the only exception to the local variables rule mentioned above.

An object can create a new place by means of $var := \mathsf{new\_pl}()$. Variable identifier var will be bound—in the store of the current object—to the new locality, which in turn will be bound—in the current allocation environment—to the physical name of the newly created object.

Similarly place loc_1 is destroyed by executing $\mathsf{del\_pl}(loc_1)$; the place will be removed from the network and all the information it contains (objects residing therein as well as the allocation environment) is lost. The short-hand $\mathsf{del\_pl}()$ destroys the place where the executing object resides.

By $\mathsf{xpt}(loc_1, loc_2, loc_3)$ locality loc_2 can be exported to the place referred to by locality loc_3 and get bound, in its allocation environment, to the place physical name which loc_1 is bound to in the allocation environment of the residence place of the executing object.

Finally, the sequential composition of action(s) Ac_1 with action(s) Ac_2 is denoted by $Ac_1; Ac_2$

3.2 Triggers

The *trigger EV t* of transition t must be a method name ($meth \in \mathcal{Z}_M$) or a method name with one parameter ($meth(x)$ with $x \in \mathsf{Par}$). The trigger has the usual pattern-matching semantics; the parameter is bound to the input value when the transition is selected for being fired. It is worth pointing out here that the UML requires the scope of a parameter to be confined to the *single* transition where it occurs as part of the trigger. The restriction to just one parameter is made only for the sake of notational simplicity.

3.3 Transition Labels and Network Specifications

The concrete syntax for the complete label of a transition t, at the UMLSC level, will be $EV\ t\ [G\ t]/AC\ t$, where the guard $[G\ t]$ is optional. The treatment of all optional parts of actions as well as short-hands is assumed to be dealt with at the static analysis level. We can now formally define *network specifications*:

Definition 1 (Network Specification). *A network specification is composed of a set* $\{H_1, \ldots, H_c\}$ *of HAs and an initialisation command* ($\mathsf{INIT}\ Ac$) *where* Ac *is a (possibly compound) action, as specified in Fig 5.*

An example of a network specification is given by the statecharts for Configurator, Server, and Agent of Figs. 2, 3, and 4 from the example in the introduction. The initialisation actions in this example are:

$$\mathsf{INIT}(l := \mathsf{new\_pl}();\ c := \mathsf{new\_ob}(\mathsf{Configurator})@l;\ c.start).$$

4 Network Semantics

The operational semantics associates a transition system to a network specification. The states of such a transition system correspond to distinct states of the network. In this section we define the semantics formally. We start with the formal definition of *stores* and *allocation environments*. Then we define *Network-* (resp. *Place-*,*Object-*) *States* and the transition relation. The definition of the latter makes use of a function for the semantic interpretation of the actions labeling statechart transitions; the remainder of the present section is devoted to the formal definition of such a function.[3]

Definition 2 (Stores). *A* store β *is a function* $\beta : \mathsf{Var} \cup \mathsf{Par} \to \mathcal{Z}_L \cup \mathcal{Z}_O \cup \{\mathsf{unbound}\}$*, where* $\mathsf{unbound} \notin \mathcal{Z}_L \cup \mathcal{Z}_O$ *is a distinguished value.*

As usual $\beta\ x = \mathsf{unbound}$ means that x is not bound by β to any value. The *empty store, unit store* and *store extension* operators ([ ], $[x \mapsto n]$, and $\triangleleft$ respectively) are defined in the usual way:

$$\begin{array}{lll} [\,]\,x & \overset{\Delta}{=} & \mathsf{unbound}, \\ & & \textbf{for all}\ \ x \in \mathsf{Var} \cup \mathsf{Par} \\ & & \\ [x \mapsto n]\ x' & \overset{\Delta}{=} & \textbf{if}\ \ x = x'\textbf{then}\ \ n\ \textbf{else}\ \mathsf{unbound}, \\ & & \textbf{for all}\ \ x, x' \in \mathsf{Var} \cup \mathsf{Par}, n \in \mathcal{Z}_L \cup \mathcal{Z}_O \\ & & \\ (\beta_1 \triangleleft \beta_2)\ x & \overset{\Delta}{=} & \textbf{if}\ \ \beta_2\ x \neq \mathsf{unbound}\ \textbf{then}\ \beta_2\ x\ \textbf{else}\ \beta_1\ x, \\ & & \textbf{for all}\ \ \textit{stores}\ \beta_1, \beta_2, x \in \mathsf{Var} \cup \mathsf{Par} \end{array}$$

Allocation environments map localities to place physical names:

Definition 3 (Allocation Environments). *An* allocation environment γ *is a function* $\gamma : \mathcal{Z}_L \cup \{\mathsf{unbound}\} \to \mathcal{Z}_P \cup \{\mathsf{unbound}\}$. *We require that* $\gamma\ \mathsf{unbound} = \mathsf{unbound}$ *for every allocation environment* γ.

As for stores, we will let [ ], respectively $[l \mapsto p]$, denote the empty, respectively unit, allocation environment, and $\gamma_1 \triangleleft \gamma_2$ denote the extension of γ_1 with γ_2, with a little bit of overloading in the notation. For each object, reserved read-only variables $\mathsf{self}, \mathsf{atLoc} \in \mathsf{Var}$ will be bound respectively to the name of the object and to the distinguished element $\mathsf{here} \in \mathcal{Z}_L$, in its current store. Similarly, for each place, here will be bound to the place physical name in its current allocation environment. Finally $\mathsf{hereafter} \in \mathcal{Z}_P$ is a distinguished place name conventionally used in the definition of the semantics of object/place destruction.

The operational semantics defines how a network may evolve as a consequence of firing transitions of the statecharts associated to the objects residing in the

[3] In the following we shall freely use a functional-like notation in our definitions where currying will be often used in function application, i.e. $f\ a_1\ a_2 \ldots\ a_n$ will be used instead of $f(a_1, a_2, \ldots, a_n)$ and function application will be considered left-associative; for function $f : X \to Y$ and $Z \subseteq X$, $f\ Z \overset{\Delta}{=} \{y \in Y \mid \exists x \in Z.\ y = fx\}$, *dom* f and *rng* f denote the *domain* and *range* of f and $f_{|Z}$ is the restriction of f to Z; in particular, $f \setminus z$ stands for $f_{|(dom\ f)\setminus\{z\}}$; for distinct $x_1, \ldots, x_n$, $f[y_1/x_1, \ldots, y_n/x_n]$ is the function which on x_j yields y_j and on any other $x' \notin \{x_1, \ldots, x_n\}$ yields $f\ x'$.

places of the network itself. We remind here that, in our approach, the primitive computational elements are the objects and that their behaviour is specified by statecharts. The evolution of objects is modelled by their internal state together with their "physical" position in the network. More specifically, at each stage of the global computation, each object resides in a specific place of the network and its internal state is composed by its current *configuration*—drawn from those of the statechart specifying its behaviour, its current local *store* —where the current values of its attributes (variables) are maintained, the current value of its *input queue*, and the current *private allocation environment.*

The evolution of the network can thus be modelled by means of a transition system where each state corresponds to a network state and each transition corresponds to a change of network state operated by firing the transitions of a STEP of the statechart of an object. In order to make the above notions more precise we need the definition below, where $\mathcal{E} \in \Theta_{\mathcal{Z}_M \cup (\mathcal{Z}_M \times (\mathcal{Z}_L \cup \mathcal{Z}_O))}$, i.e. the elements of input queues are method invocations, with possibly one parameter. For $(m, n) \in \mathcal{Z}_M \times (\mathcal{Z}_L \cup \mathcal{Z}_O)$, we use the more common, "constructor-like", syntax $m(n)$.

Definition 4 (Network, Place and Object States). *A* network state N *is a finite set of place states. A* place state P *of a network state* N *is a triple* $(p, \gamma, \mu) \in N$ *where* $p \in \mathcal{Z}_P \setminus \{\mathsf{hereafter}\}$ *is the physical name for* P*, and is required to be unique net-wide;* γ *is the allocation environment of* P *and* μ *is the finite set of object states of* P*. An object state* O *of place state* $P = (p, \gamma, \mu)$ *of network state* N *models the state of an object and is a 6-tuple* $(o, H, \lambda, \mathcal{C}, \beta, \mathcal{E})$ *where* $o \in \mathcal{Z}_O^p$ *is the name of the object and is required to be unique network-wide;* H *is the reference to the HA which specifies the behaviour of* o *and* λ *(respectively* $\mathcal{C}, \beta, \mathcal{E}$*) is the current private allocation environment (respectively configuration, store, input queue) of* o*.*

It is worth pointing out here that as an obvious consequence of place name uniqueness within a network state N the latter can be used and manipulated as a (finite domain) function on $\mathcal{Z}_P$ such that $p \in dom\ N$ and $N\ p = (\gamma, \mu)$ if, and only if $(p, \gamma, \mu) \in N$. Similarly, also μ can be used and manipulated as a (finite domain) function on $\mathcal{Z}_O^p$ with $o \in dom\ \mu$ and $\mu\ o = (H, \lambda, \mathcal{C}, \beta, \mathcal{E})$ if, and only if $(o, H, \lambda, \mathcal{C}, \beta, \mathcal{E}) \in \mu$. In the sequel, we assume $\{\mathsf{unbound}, \mathsf{hereafter}\} \cap (dom\ N) = \emptyset$ for all network states N and $\mathsf{unbound} \notin (dom\ \mu)$ for all place states (p, γ, μ).

The operational semantics associates a transition system $(S, \longrightarrow, S_0)$ to a network specification. S is the set of states of such transition system, which are network states, defined as above. A distinguished network state corresponds to the initial state S_0. Conventionally, such a state consists of the single place $(\mathsf{init\_pl}, [\,], \{(\mathsf{init\_o}, \mathsf{INITHA}, [\,], \{1\}, [\,], \langle \mathsf{init\_ev} \rangle)\})$ where INITHA is a conventional HA composed of a single state—1— which is source and target of a single transition labelled by $\mathsf{init\_ev}/as$ where as is the argument of the INIT initialisation command.[4] The transition relation $\longrightarrow$ is defined by means of a logical deduc-

[4] It is assumed that init_o and init_ev do not occur in any network specification.

tion system, and the definition is given in two stages: the *Top Rule* and the *Core Semantics*. The Top Rule is shown in Fig. 6 and in turn uses the Core Semantics, which has been introduced in Sect. 2. The Top Rule stipulates that in order for the network to evolve from network state N to state N' there must exist an object o (2nd premise) in a place p of the current network state (1st premise), the statechart of which—H—can perform a STEP from the (non-error) configuration (3rd premise) $\mathcal{C}$ to $\mathcal{C}'$ (5th premise) when event ev is selected from its input queue (5th premise). The STEP transition generated by the Core Semantics is labelled by the set L of the HA transitions which are fired in the STEP, the pair (ev, β), where β is the current store of o, and the pair (Ac, ξ) where Ac is a sequence of actions—the actions of the transitions in L—and ξ is the set of bindings of the parameters occurring in the triggers of such transitions to the input event ev. Ac is a *symbolic* representation of transition actions; thus we need to interpret them. This is achieved by means of the interpretation function $\mathcal{N}$ which actually computes the new network state N' (6th premise). N' is computed by applying $\mathcal{N}[\![\mathsf{Ac}]\!]$ to a network state which is the same as N except that the new configuration $\mathcal{C}'$ and the remaining input queue $\mathcal{E}'$ are recorded for object o. Notice that the store of o is not updated (yet) since the new store will be (part of) the result of the execution of Ac.

$$\frac{\begin{array}{l}(p,\gamma,\mu) \in N\\ (o,H,\lambda,\mathcal{C},\beta,\mathcal{E}) \in \mu\\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}}\\ \mathsf{Sel}\ \mathcal{E}\ \ ev\ \mathcal{E}'\\ H :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_{L} \mathcal{C}'\\ N' = \mathcal{N}[\![\mathsf{Ac}]\!]\ (N[(\gamma,\mu[(H,\lambda,\mathcal{C}',\beta,\mathcal{E}']/o)/p],p)\ o\ \xi\end{array}}{N \longrightarrow N'}$$

Fig. 6. Transition relation definition (top rule)

In the following we define the action interpretation function $\mathcal{N}$. Actually $\mathcal{N}$ simply returns the first element of the pair resulting from the application of function $\mathcal{I}$ to the same arguments.

$$\mathcal{N}[\![Ac]\!]\ (N,p)\ o\ \xi \ \ \overset{\Delta}{=}\ \ N' \text{ where } (N',p') = \mathcal{I}[\![Ac]\!]\ (N,p)\ o\ \xi$$

The definition of $\mathcal{I}$ uses the following auxiliary functions.

$$\begin{array}{lll}\mathsf{ERR}\ N\ p\ o & \overset{\Delta}{=} & N[(\gamma,\mu[(nil,[],\mathcal{C}_{\mathsf{err}},[],\langle\rangle)/o])/p],\\ & & \textbf{for all}\ \ net.\ states\ N, p \in dom\ N, o \in \mathcal{Z}_O\end{array}$$

$\mathsf{ERR}\ N\ p\ o$ is the network state which differs from N only because the *erratic* object state $(o, nil, [], \mathcal{C}_{\mathsf{err}}, [], \langle\rangle)$ is present in place state p.

Function $\mathcal{V}$ is defined in the usual way.

$\mathcal{V}[\![exp]\!]\ \beta \quad \stackrel{\Delta}{=}$ **if** $exp \in \mathsf{Var} \cup \mathsf{Par}$
then $\beta\ exp$
else exp,
for all *stores* β,
$exp \in \mathsf{Var} \cup \mathsf{Par} \cup \mathcal{Z}_L$

$\mathsf{new}\ \mathcal{Z}_P \quad \stackrel{\Delta}{=}$ *a* fresh new *place physical name* $p \in \mathcal{Z}_P \setminus \{\mathsf{init\_pl}, \mathsf{hereafter}\}$ *different from any place physical name already generated.*

$\mathsf{new}\ \mathcal{Z}_L \quad \stackrel{\Delta}{=}$ *a* fresh new *locality name* $l \in \mathcal{Z}_L \setminus \{\mathsf{here}\}$ *different from any locality textually occurring in the network specification or already generated.*

$\mathsf{new}\ \mathcal{Z}_O^p \quad \stackrel{\Delta}{=}$ *a* fresh new *object name* $o \in \mathcal{Z}_O^p \setminus \{\mathsf{init\_o}\}$ *textually different from any object name occurring in the network specification or already generated.*

For action Ac, $\mathcal{I}[\![\mathsf{Ac}]\!]$ is a function which takes a pair (N, p)—where N is a network state and $p \in \mathcal{Z}_P$—an object name $o \in \mathcal{Z}_O$ and a parameter binding ξ. $\mathcal{I}[\![\mathsf{Ac}]\!]\ (N, p)\ o\ \xi$ is a pair (N', p') where N' is the network state resulting from the execution of the actions Ac and $p' \in \mathcal{Z}_P$. $\mathcal{I}[\![\mathsf{Ac}]\!]$ is defined by induction on the structure of Ac. In all cases it is first of all required that $p \in dom\ N$, i.e. $\exists \gamma, \mu.\ (\gamma, \mu) = N\ p$ and that o is not erratic, with $o \in dom\ \mu$.

In the following, we list all cases of the definition, together with a short informal explanation, when necessary.

$\mathcal{I}[\![var := exp]\!]\ (N, p)\ o\ \xi \quad \stackrel{\Delta}{=}$
if $\gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, n$
exist such that
$(\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}},$
$n = \mathcal{V}[\![exp]\!]\ (\beta \triangleleft \xi),\ n \neq \mathsf{unbound}$
then $N[(\gamma, \mu[(H, \lambda, \mathcal{C}, \beta[n/var], \mathcal{E})/o])/p], p)$
else $(\mathsf{ERR}\ N\ p\ o, p)$

Access to an uninitialised variable exp (i.e. $\mathcal{V}[\![exp]\!]\ (\beta \triangleleft \xi) = \mathsf{unbound}$) in an assignment action $var := exp$ brings o to the erratic state, otherwise var is bound in the store to the value n of exp. Notice that expressions are evaluated—using function $\mathcal{V}$—in the current store *temporarily extended* with the parameter bindings.

$\mathcal{I}[\![obj@loc.meth(exp)]\!]\ (N, p)\ o\ \xi \quad \stackrel{\Delta}{=}$
if $\gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, l, o', p', \gamma', \mu', H', \lambda', \mathcal{C}', \beta', \mathcal{E}', m, n$
exist such that
$(\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}},$

$o' = \mathcal{V}[\![obj]\!]\ (\beta \lhd \xi),\ l = \mathcal{V}[\![loc]\!]\ (\beta \lhd \xi),\ p' = (\gamma \lhd \lambda)\ l,$
$(\gamma', \mu') = N\ p',\ (H', \lambda', \mathcal{C}', \beta', \mathcal{E}') = \mu'\ o',$
$n = \mathcal{V}[\![exp]\!]\ (\beta \lhd \xi)$
then $(N[(\gamma', \mu'[(H', \lambda', \mathcal{C}', \beta', \mathsf{Add}\ meth(n)\ \mathcal{E}')/o'])/p'], p)$
else $(\mathsf{ERR}\ N\ p\ o, p)$

In the execution of sending an asynchronous call $meth(exp)$ to object $obj@loc$, performed by object o residing in place p of N, it is required that locality $\mathcal{V}[\![loc]\!]\ (\beta \lhd \xi) = l$ is bound, in the local allocation environment γ (possibly extended with the private allocation environment of o if any) to the physical name p' of a place where an object named $\mathcal{V}[\![obj]\!]\ (\beta \lhd \xi) = o'$ exists. In this case, $meth(\mathcal{V}[\![exp]\!]\ (\beta \lhd \xi))$ is added in the input queue of o'; otherwise o ends up in the erratic state.

$\mathcal{I}[\![\mathsf{mv\_ob}(obj@loc_1, loc_2)]\!]\ (N, p)\ o\ \xi \ \stackrel{\Delta}{=}$
if $\gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, l_1, l_2, p_1, p_2, \gamma_1, \gamma_2, \mu_1, \mu_2, o', p'$
exist such that
$(\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}},$
$l_1 = \mathcal{V}[\![loc_1]\!]\ (\beta \lhd \xi),\ p_1 = (\gamma \lhd \lambda)\ l_1,\ (\gamma_1, \mu_1) = N\ p_1,$
$l_2 = \mathcal{V}[\![loc_2]\!]\ (\beta \lhd \xi),\ p_2 = (\gamma \lhd \lambda)\ l_2,\ (\gamma_2, \mu_2) = N\ p_2,$
$o' = \mathcal{V}[\![obj]\!]\ (\beta \lhd \xi),\ o' \in (dom\ \mu_1),$
$p' =$ **if** $p = p_1\ , o = o'$ **then** p_2 **else** p
then $(N[(\gamma_1, \mu_1 \setminus o')/p_1, (\gamma_2, \mu_2[\mu_1\ o'/o'])/p_2], p')$
else $(\mathsf{ERR}\ N\ p\ o, p)$

$\mathcal{I}[\![\mathsf{mv\_cl}(obj@loc_1, loc_2)]\!]\ (N, p)\ o\ \xi \ \stackrel{\Delta}{=}$
if $\gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, l_1, l_2, p_1, p_2, \gamma_1, \gamma_2, \mu_1, \mu_2, \mu'_2 o', p',$
$H', \lambda', \mathcal{C}', \beta', \mathcal{E}'$
exist such that
$(\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}},$
$l_1 = \mathcal{V}[\![loc_1]\!]\ (\beta \lhd \xi),\ p_1 = (\gamma \lhd \lambda)\ l_1,\ (\gamma_1, \mu_1) = N\ p_1,$
$l_2 = \mathcal{V}[\![loc_2]\!]\ (\beta \lhd \xi),\ p_2 = (\gamma \lhd \lambda)\ l_2,\ (\gamma_2, \mu_2) = N\ p_2,$
$o' = \mathcal{V}[\![obj]\!](\beta \lhd \xi),\ o' \in (dom\ \mu_1),$
$(H', \lambda', \mathcal{C}', \beta', \mathcal{E}') = \mu'\ o',$
$\mu'_2 = \mu_2[(H', \gamma_1 \ \lhd\ \lambda', \mathcal{C}', \beta', \mathcal{E}')/o']$
then $(N[(\gamma_1, \mu_1 \setminus o')/p_1, (\gamma_2, \mu'_2)/p_2], p')$
else $(\mathsf{ERR}\ N\ p\ o, p)$

The successful execution of $\mathsf{mv\_ob}(obj@loc_1, loc_2)$ requires that the object denoted by obj resides in the place p_1 referred to by loc_1 *and* that there is no object with the same name in the place p_2 referred to by loc_2. If this is the case, this object is removed from the residence place p_1 and added to the set of objects of p_2, i.e. is moved from p_1 to p_2. In the case of $\mathsf{mv\_cl}$, furthermore the current allocation environment γ_1 of p_1 is added to the (possibly empty) private allocation environment of the object. Notice that actions $\mathsf{mv\_ob}$ and $\mathsf{mv\_cl}$ can be applied also by o to itself, when obj evaluates to o and loc_1 refers to p. In this case the residence place of o changes to the value of loc_2. We keep track of the residence of o as it results from the execution of Ac in the second element of the result of $\mathcal{I}[\![\mathit{Ac}]\!]\ (N,p)\ o\ \xi$. The reader is invited to check how this information is dealt with in the definition of $\mathcal{I}[\![Ac_1; Ac_2]\!]$.

```
$\mathcal{I}[\![var := \mathsf{new\_ob}(H_0, \mathcal{C}_0, \beta_0, \mathcal{E}_0)@loc]\!]\ (N,p)\ o\ \xi\ \stackrel{\Delta}{=}$
   **if** $\gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, l, o', p', \gamma', \mu' O, O'$
   **exist such that**
   $(\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}}$
   $l = \mathcal{V}[\![loc]\!]\ (\beta \lhd \xi),\ p' = (\gamma \lhd \lambda)\ l,\ (\gamma', \mu') = N\ p',$
   $o' = \mathsf{new}\ \mathcal{Z}_O^{p'},$
   $O = (H, \lambda, \mathcal{C}, \beta[o'/var], \mathcal{E}),$
   $O' = (H_0, [], \mathcal{C}_0, \beta_0[o'/\mathsf{self}, \mathsf{here}/\mathsf{atLoc}], \mathcal{E}_0)$
   **then if** $p = p'$
         **then** $(N[(\gamma, \mu[O/o, O'/o'])/p], p)$
         **else** $(N[(\gamma, \mu[O/o])/p, (\gamma', \mu[O'/o'])/p'], p)$
   **else** $(\mathsf{ERR}\ N\ p\ o, p)$
```

```
$\mathcal{I}[\![var := \mathsf{new\_cl}(H_0, \mathcal{C}_0, \beta_0, \mathcal{E}_0)@loc]\!]\ (N,p)\ o\ \xi\ \stackrel{\Delta}{=}$
   **if** $\gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, l, o', p', \gamma', \mu', O, O'$
   **exist such that**
   $(\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}},$
   $l = \mathcal{V}[\![loc]\!]\ (\beta \lhd \xi),\ p' = (\gamma \lhd \lambda)\ l,\ (\gamma', \mu') = N\ p',$
   $o' = \mathsf{new}\ \mathcal{Z}_O^{p'},$
   $O = (H, \lambda, \mathcal{C}, \beta[o'/var], \mathcal{E}),$
   $O' = (H_0, \gamma, \mathcal{C}_0, \beta_0[o'/\mathsf{self}, \mathsf{here}/\mathsf{atLoc}], \mathcal{E}_0)$
   **then if** $p = p'$
         **then** $(N[(\gamma, \mu[O/o, O'/o'])/p], p)$
         **else** $(N[(\gamma, \mu[O/o])/p, (\gamma', \mu[O'/o'])/p'], p)$
   **else** $(\mathsf{ERR}\ N\ p\ o, p)$
```

The creation $\mathsf{new\_ob}$ of a new object in an *existing* place referred to by loc requires the modification of the store of the creator object o in order to bind variable var to the name o' of the newly created object. Moreover the new object is placed in the place referred to by loc. Notice that the initial store of the new object binds atLoc to here and self to o'. In the case of $\mathsf{new\_cl}$, the private allocation environment of the newly created object is initialised to the allocation environment of the place where the creator resides.

```
$\mathcal{I}[\![\mathsf{del\_ob}(obj@loc)]\!]\ (N,p)\ o\ \xi\ \stackrel{\Delta}{=}$
   **if** $\gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, l_1, p_1, \gamma_1, \mu_1, o', p'$
   **exist such that**
   $(\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}},$
   $l_1 = \mathcal{V}[\![loc]\!]\ (\beta \lhd \xi),\ p_1 = (\gamma \lhd \lambda)\ l_1,\ (\gamma_1, \mu_1) = N\ p_1,$
   $o' = \mathcal{V}[\![obj]\!]\ (\beta \lhd \xi),$
   $p' = \textbf{if}\ \ p = p_1,\ o = o'\ \textbf{then}\ \ \mathsf{hereafter}\ \textbf{else}\ \ p$
```

```
  then $(N[(\gamma, \mu[(H, \lambda, \mathcal{C}, \beta \setminus obj, \mathcal{E})/o])/p, (\gamma_1, \mu_1 \setminus o')/p_1], p')$
  else $(\mathsf{ERR}\ N\ p\ o, p)$
```

An existing object *obj* residing in a place referred to by *loc* is destroyed by executing del_ob(*obj*@*loc*). The result will be that the object will be removed from the set of objects of *loc* and variable *obj* will be unbound after del_ob will have been executed. Notice that if *obj*@*loc* is exactly the object which is executing the action then its residence place changes to hereafter (we remind the reader that hereafter $\in$ (*dom* N) for *no* network state N). Notice moreover that the semantics is undefined if action del_ob occurs in *Ac* but *not* as its *last* element.

```
$\mathcal{I}[\![var := \mathsf{new\_pl}()]\!]\ (N, p)\ o\ \xi \ \ \stackrel{\Delta}{=}$
  if $\gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, O, l, p'$
  exist such that
  $(\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}},$
  $l = \mathsf{new}\ \mathcal{Z}_L,\ p' = \mathsf{new}\ \mathcal{Z}_P,$
  $O = (H, \lambda, \mathcal{C}, \beta[l/var], \mathcal{E})$
  then $(N[(\gamma[p'/l], \mu[O/o])/p, ([\mathsf{here} \mapsto p'], \emptyset)/p'], p)$
  else $(\mathsf{ERR}\ N\ p\ o, p)$
```

The successful execution of *var* := new_pl() creates a new place where here is bound to its physical name by its (otherwise empty) allocation environment. Moreover, a new locality is bound to such physical name in the allocation environment of the place where the executing object resides and such locality is bound to variable *var* in the store of the object.

```
$\mathcal{I}[\![\mathsf{del\_pl}(loc)]\!]\ (N, p)\ o\ \xi \ \ \stackrel{\Delta}{=}$
  if $\gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, l, p', p''$
  exist such that
  $(\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}}$
  $l = \mathcal{V}[\![loc]\!]\ (\beta \triangleleft \xi),\ p'' = (\gamma \triangleleft \lambda),$
  $p' =$ if $\ p = p''$ then $\ \mathsf{hereafter}$ else $\ p$
  then $((N[(\gamma \setminus l, \mu[(H, \lambda, \mathcal{C}, \beta \setminus loc, \mathcal{E})/o])/p]) \setminus p'', p')$
  else $(\mathsf{ERR}\ N\ p\ o, p)$
```

The interpretation of place destruction (del_pl) should be clear to the reader; the only exception is when the object executing it destroys the place where it resides. Notice that the semantics is undefined if action del_pl occurs in *Ac* but *not* as its *last* element.

```
$\mathcal{I}[\![\mathsf{xpt}(loc_1, loc_2, loc_3]\!]\ (N, p)\ o\ \xi \ \ \stackrel{\Delta}{=}$
  if $\gamma, \mu, H, \lambda, \mathcal{C}, \beta, \mathcal{E}, l_1, l_2, l_3, p_1, p_3, \gamma_3, \mu_3$
  exist such that
  $(\gamma, \mu) = N\ p,\ (H, \lambda, \mathcal{C}, \beta, \mathcal{E}) = \mu\ o,\ \mathcal{C} \neq \mathcal{C}_{\mathsf{err}}$
  $l_1 = \mathcal{V}[\![loc_1]\!]\ (\beta \triangleleft \xi),\ p_1 = (\gamma \triangleleft \lambda)\ l_1,$
  $l_2 = \mathcal{V}[\![loc_2]\!]\ (\beta \triangleleft \xi),$
```

$l_3 = \mathcal{V}[\![loc_3]\!]\ (\beta \triangleleft \xi),\ p_3 = (\gamma \triangleleft \lambda)\ l_3,$
$(\gamma_3, \mu_3) = N\ p_3,\ l_2 \notin dom\ \gamma_3$
then $(N[(\gamma_3[p_1/l_2], \mu_3)/p_3], p)$
else $(\mathsf{ERR}\ N\ p\ o, p)$

$$\mathcal{I}[\![Ac_1; Ac_2]\!]\ (N,p)\ o\ \xi \ \stackrel{\Delta}{=}$$
$$\quad \mathcal{I}[\![Ac_2]\!]\ (\mathcal{I}[\![Ac_1]\!]\ (N,p)\ o\ \xi)\ o\ \xi$$

The semantics of xpt and sequentialization is self-explanatory.

5 Conclusions

In this paper, UML statecharts have been extended with a notion of mobility. In particular *mobile computation*, where computational units migrate from one node to another within a network has been considered, as opposed to *mobile computing*, which addresses dynamic communication structures [12].

A formal operational semantics for the extended notation has been provided which covers all major aspects of UML statecharts—like state hierarchy, interlevel transitions, a parametric treatment of transition priority and input queue, intra- and inter- statechart concurrency, and run-to-completion. Furthermore, it includes dynamic object management, i.e. object creation and object destruction, for objects (the behaviour of which is) specified by statecharts; finally notions specific to dynamic network management are addressed: network places, network architecture management, and mobility (i.e. object migration).

An example of a model of a network service which exploits mobility for resource usage balance has been provided using our mobile extension of UMLSCs.

We are not aware of any proposal in the literature which combines all the above mentioned issues in a single formal framework which is moreover completely compatible and consistent with other "views" and extensions of UML statecharts, like testing theories, stochastic behaviour modelling, and analysis and LTL/BTL model-checking.[5]

The space of network places is *flat*, which is similar to that of KLAIM [5]. Other proposals, like for instance [4, 10], assume a hierarchical structure for the place space. We chose a flat approach mainly for the sake of simplicity. A hierarchical place structure would open the way toward mobility of *places* and, since mobility is part of computation and objects are the computational units, this would bring to the blurring of the conceptual difference between objects and places. In addition, the conceptual difference between physical names and localities would need to be revisited. We leave all the above issues for further study.

The impact of our extension on the UML meta model—e.g. what additional meta classes are needed and whether these can be introduced by stereotypes or not—is also left for future study.

[5] Papers describing the above mentioned issues can be found at: http://fmt.isti.cnr.it.

Another issue for further study is the integration of the mobile computing approach proposed in [12] with the mobile computation one proposed in the present paper, as well as the interplay between mobile computing and mobile computation in a framework where also place mobility is considered, as briefly mentioned above. Finally, we are interested in developing useful theories for the extension we proposed in the present paper, like, e.g. access control and security, in a similar way as in [5].

References

1. B. Bauer, J. Müller, and J. Odell. Agent UML: A Formalism for Specifying Multiagent Interaction. In P. Ciancarini and M. Wooldridge, editors, *Agent-Oriented Software Engineering*, volume 1957 of *Lecture Notes in Computer Science*, pages 91–103. Springer-Verlag, 2001.
2. H. Baumeister, N. Koch, P. Kosiuczenko, and M. Wirsing. Extending Activity Diagrams to Model Mobile Systems. In M. Aksit, M. Mezini, and R. Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World.*, volume 2591 of *Lecture Notes in Computer Science*, pages 278–293. Springer-Verlag, 2002.
3. J. Broersen and R. Wieringa. Interpreting UML-statecharts in a modal μ-calculus. Unpublished manuscript, 1997.
4. L. Cardelli and A. Gordon. Mobile ambients. In M. Nivat, editor, *FoSSaCS'98*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–145. Springer-Verlag, 1998.
5. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A kernel language for agents interaction and mobility. *IEEE Transactions on Software Engineering*, 24(5):315–329, 1998.
6. S. Gnesi, D. Latella, and M. Massink. Modular semantics for a UML Statechart Diagrams kernel and its extension to Multicharts and Branching Time Model Checking. *The Journal of Logic and Algebraic Programming. Elsevier Science*, 51(1):43–75, 2002.
7. C. Klein, A. Rausch, M. Sihling, and Z. Wen. Extension of the Unified Modeling Language for mobile agents. In K. Siau and T. Halpin, editors, *Unified Modeling Language: Systems Analysis, Design and Development Issues*, chapter 8. Idea Group Publishing, Hershey, PA and London, 2001.
8. Alexander Knapp, Stephan Merz, and Martin Wirsing. On refinement of mobile UML state machines, 2004. to appear in Proc. AMAST 2004.
9. P. Kosiuczenko. Sequence Diagrams for Mobility. In J. Krogstie, editor, *MobIMod 2002*, volume XXXX of *Lecture Notes in Computer Science*. Springer-Verlag, 2003. (To appear).
10. A. Kuhn and von Oheimb D. Interacting state machines for mobility. In K. Araki, S. Gnesi, and D. Mandrioli, editors, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 698–718. Springer-Verlag, 2003.
11. D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. In P. Ciancarini, A. Fantechi, and R. Gorrieri, editors, *IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Oriented Distributed Systems*, pages 331–347. Kluwer Academic Publishers, 1999. ISBN 0-7923-8429-6.

12. D. Latella and M. Massink. On mobility extensions of UML Statecharts; a pragmatic approach. In E. Najm, U. Nestmann, and P. Stevens, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 2884 of *Lecture Notes in Computer Science*, pages 199–213. Springer-Verlag, 2003.
13. J. Lilius and I. Paltor Porres. Formalising UML state machines for model checking. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard.*, volume 1723 of *Lecture Notes in Computer Science*, pages 430–445. Springer-Verlag, 1999.
14. J. Lilius and I. Paltor Porres. The semantics of UML state machines. Technical Report 273, Turku Centre for Computer Science, 1999.
15. S. Merz, M. Wirsing, and J. Zappe. A Spatio-Temporal Logic for the Specification and Refinement of Mobile Systems. In M. Pezzé, editor, *Fundamental Approaches to Software Engineering (FASE 2003)*, volume 2621 of *Lecture Notes in Computer Science*. Springer-Verlag, 2003.
16. E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In R. Shyamasundar and K. Euda, editors, *Third Asian Computing Science Conference. Advances in Computing Sience - ASIAN'97*, volume 1345 of *Lecture Notes in Computer Science*, pages 181–196. Springer-Verlag, 1997.
17. Object Management Group, Inc. *OMG Unified Modeling Language Specification - version 1.5*, 2003. http://www.omg.org/cgi-bin/doc?formal/03-03-01.
18. M. von der Beeck. A structured operational semantics for UML-statecharts. *Software Systems Modeling. Springer*, (1):130–141, 2002.
19. R. Wieringa and J. Broersen. A minimal transition system semantics for lightweight class and behavior diagrams. In M. Broy, D. Coleman, T. Maibaum, and B. Rumpe, editors, *Proceedings of the ICSE98 Workshop on Precise Semantics for Software Modeling techniques*, 1998.

A Hierarchical Automata

In this section we informally recall some basic notions related to UMLSCs. They are treated in depth in [11, 6]. We use hierarchical automata (HAs) [16] as the abstract syntax for UMLSCs. HAs are composed of simple sequential automata related by a *refinement function.* In [11] an algorithm for mapping a UMLSC to a HA is given. Here we just recall the main ingredients of this mapping, by means of a simple example. Consider the UMLSC of Fig. 7 (). Its HA is shown top on the bottom of the figure. Roughly speaking, each OR-state of the UMLSC is mapped into a sequential automaton of the HA while basic and AND-states are mapped into states of the sequential automaton corresponding to the OR-state immediately containing them. Moreover, a refinement function maps each state in the HA corresponding to an AND-state into the set of the sequential automata corresponding to its component OR-states. In our example (Fig. 7, bottom), OR-states $s0, s4, s5$ and $s7$ are mapped to sequential automata $A0, A1, A2$ and $A3$, while state $s1$ of $A0$, corresponding to AND-state $s1$ of our UMLSC, is refined into $\{A1, A2\}$. Non-interlevel transitions are represented in the obvious way: for instance transition $t8$ of the HA represents the transition from state $s8$ to state $s9$ of the UMLSC. The labels of transitions are collected in Table 1; for example the *trigger* of $t8$, namely $EV\ t8$, is $e2$ while its associated *action*, namely $AC\ t8$

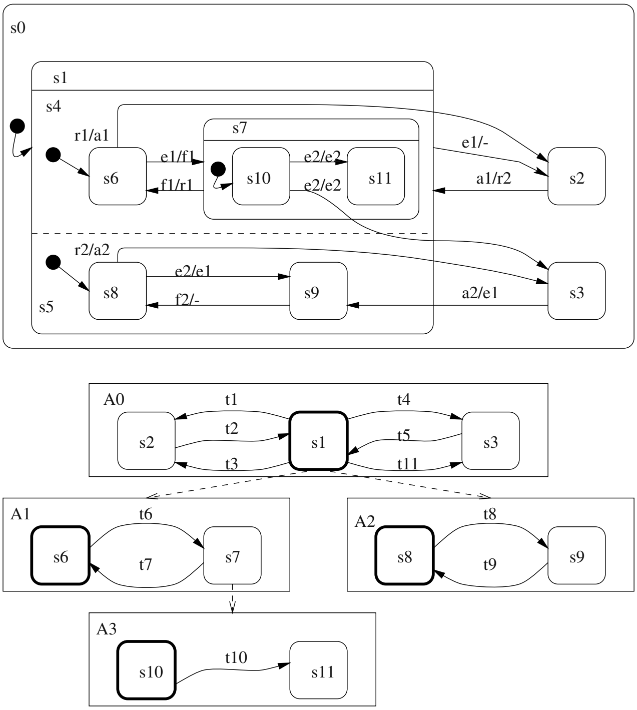


Fig. 7. A UMLSC and its HA

Table 1. Transition labels for the HA of Fig. 7

t	$SR\,t$	$EV\,t$	$AC\,t$	$TD\,t$
$t1$	$\{s6\}$	$r1$	$a1$	$\emptyset$
$t2$	$\emptyset$	$a1$	$r2$	$\{s6, s8\}$
$t3$	$\emptyset$	$e1$	ϵ	$\emptyset$
$t4$	$\{s8\}$	$r2$	$a2$	$\emptyset$
$t5$	$\emptyset$	$a2$	$e1$	$\{s6, s9\}$
$t6$	$\emptyset$	$e1$	$f1$	$\{s10\}$
$t7$	$\emptyset$	$f1$	$r1$	$\emptyset$
$t8$	$\emptyset$	$e2$	$e1$	$\emptyset$
$t9$	$\emptyset$	$f2$	ϵ	$\emptyset$
$t10$	$\emptyset$	$e2$	$e2$	$\emptyset$
$t11$	$\{s10\}$	$e2$	$e2$	$\emptyset$

consists in $e1$. Label $e2$ can model the activation of a method of an object the behaviour of which is modeled by the statechart and, respectively, $e1$ can be the invocation of a method, which takes place if and when $t8$ is fired.

An interlevel transition is represented as a transition t departing from (the HA state corresponding to) its highest source and pointing to (the HA state corresponding to) its highest target. The set of the other sources, resp., targets, are recorded in the *source restriction*—$SR\ t$, resp. *target determinator* $TD\ t$, of t. So, for instance, $SR\ t1 = \{s6\}$ means that a necessary condition for $t1$ to be enabled is that the current state configuration contains not only $s1$ (the source of $t1$), but *also* $s6$. Similarly, when firing $t2$ the new state configuration will contain $s6$ and $s8$, besides $s1$. Finally, each transition has a guard $G\ t$, not shown in this example.

Transitions originating from the same state are said to be in *conflict*. The notion of *conflict* between transitions is to be extended in order to deal with state hierarchy and a priority notion between conflicting transition is defined. When transitions t and t' are in conflict we write $t\#t'$. Intuitively transitions coming from deeper states have higher priority. For the purposes of the present paper it is sufficient to say that priorities form a partial order. We let πt denote the priority of transition t and $\pi t \sqsubseteq \pi t'$ mean that t has lower priority than (the same priority as) t'.

In the sequel we will be concerned only with HAs. In particular, for a given network specification, we shall make reference to the set $\{H_1, \ldots, H_c\}$ of the HAs associated to the UMLSCs $SC_1, \ldots, SC_c$ used in the specification.

A.1 Basic Definitions

The first notion we need to define is that of (sequential) automaton:[6]

Definition 5 (Sequential Automata). *A sequential automaton A is a 4-tuple $(\sigma_A, s^0_A, \lambda_A, \delta_A)$ where σ_A is a finite set of* states *with $s^0_A \in \sigma_A$ the* initial state

[6] In the following we shall freely use a functional-like notation in our definitions where: (i) currying will often be used in function application, i.e. $f\ a_1\ a_2 \ldots\ a_n$ will be used instead of $f(a_1, a_2, \ldots, a_n)$ and function application will be considered left-associative; (ii) for function $f : X \to Y$ and $Z \subseteq X$, $f\ Z \stackrel{\Delta}{=} \{y \in Y \mid \exists x \in Z.\ y = fx\}$, $dom\ f$ and $rng\ f$ denote the *domain* and *range* of f and $f_{|Z}$ is the restriction of f to Z; in particular, $f \backslash z$ stands for $f_{|(dom\ f) \backslash \{z\}}$; for distinct $x_1, \ldots, x_n$, $f[y_1/x_1, \ldots, y_n/x_n]$ is the function which on x_j yields y_j and on any other $x' \notin \{x_1, \ldots, x_n\}$ yields $f\ x'$; for functions f and g such that for all $x \in (dom\ f \cap dom\ g)$ $f\ x = g\ x$ holds we will often let $f \cup g$ denote the function which yields $f\ x$ if $x \in dom\ f$ and $g\ x$ if $x \in dom\ g$ and we extend the notation to n functions in the obvious way. By $\exists_1 x.\ P\ x$ we mean "there exists a unique x such that $P\ x$". Finally, for set D, we let D^* denote the set of finite sequences over D. The empty sequence will be denoted by ϵ and, for $d \in D$, with a bit of overloading, we will often use d also for the unit sequence containing only d; the concatenation of sequence x with sequence y will be indicated by xy. For sequences x, y and z we let predicate $\mathsf{mrg}\ x\ y\ z$ hold if, and only if z is a non-deterministic merge (or interleaving) of x and y, that is z is a permutation of xy such that the occurrence order in x (respectively y) of the elements of x (respectively y) is preserved in z; a possible definition for mrg is $\mathsf{mrg}\ x\ y\ z \stackrel{\Delta}{=} \exists w \in (D \times \{1,2\})^*.\ pr1\ w = z \wedge pr1\ (only\ 1\ w) = x \wedge pr1\ (only\ 2\ w) = y$ where $pr1\ \epsilon \stackrel{\Delta}{=} \epsilon$, $pr1\ (e,j)l \stackrel{\Delta}{=} e(pr1\ l)$, $only\ j\ \epsilon \stackrel{\Delta}{=} \epsilon$, and $only\ j\ (e,j')l \stackrel{\Delta}{=} (\textbf{if}\ \ j = j'\ \textbf{then}\ \ (e,j)\ \textbf{else}\ \epsilon)(only\ j\ l)$; the extension of mrg to n sequences, $\mathsf{mrg}^n_{j=1}\ x_j\ z$, is defined in the obvious way.

λ_A *is a finite set of* transition labels *and* $\delta_A \subseteq \sigma_A \times \lambda_A \times \sigma_A$ *is the* transition relation.

We assume that all transitions are uniquely identifiable. This can be easily achieved by just assigning them arbitrary unique names, as we shall do throughout this paper. For sequential automaton A let functions $SRC, TGT : \delta_A \rightarrow \sigma_A$ be defined as $SRC(s, l, s') = s$ and $TGT(s, l, s') = s'$.

HAs are defined as follows:

Definition 6 (Hierarchical Automata). *A HA H is a 3-tuple (F, E, ρ) where F is a finite set of sequential automata with mutually disjoint sets of states, i.e. $\forall A_1, A_2 \in F.\ \sigma_{A_1} \cap \sigma_{A_2} = \emptyset$ and E is a finite set of transition labels; the* refinement function $\rho : \bigcup_{A \in F} \sigma_A \rightarrow 2^F$ *imposes a tree structure to F, i.e. (i) there exists a unique root automaton $A_{root} \in F$ such that $A_{root} \notin \bigcup rng\ \rho$, (ii) every non-root automaton has exactly one ancestor state: $\bigcup rng\ \rho = F \setminus \{A_{root}\}$ and $\forall A \in F \setminus \{A_{root}\}.\ \exists_1 s \in \bigcup_{A' \in F \setminus \{A\}} \sigma_{A'}.\ A \in (\rho\ s)$ and (iii) there are no cycles: $\forall S \subseteq \bigcup_{A \in F} \sigma_A.\ \exists s \in S.\ S \cap \bigcup_{A \in \rho s} \sigma_A = \emptyset$.*

We say that a state s for which $\rho\ s = \emptyset$ holds is a *basic* state. Every sequential automaton $A \in F$ characterises a HA in its turn: intuitively, such a HA is composed by all those sequential automata which lay below A, including A itself, and has a refinement function ρ_A which is a restriction of ρ:

Definition 7. *For $A \in F$ the* automata *and* states under A *are defined respectively as*
$\mathcal{A}\ A \stackrel{\Delta}{=} \{A\} \cup (\bigcup_{A' \in \left(\bigcup_{s \in \sigma_A} (\rho_A s)\right)} (\mathcal{A}\ A'))$, $\mathcal{S}\ A \stackrel{\Delta}{=} \bigcup_{A' \in \mathcal{A}\ A} \sigma_{A'}$

The definition of sub-hierarchical automaton follows:

Definition 8 (Sub-hierarchical Automata). *For $A \in F$, (F_A, E, ρ_A), where $F_A \stackrel{\Delta}{=} (\mathcal{A}\ A)$, and $\rho_A \stackrel{\Delta}{=} \rho_{|(\mathcal{S}\ A)}$, is the HA characterised by A.*

In the sequel for $A \in F$ we shall refer to A both as a sequential automaton and as the sub-hierarchical automaton of H it characterises, the role being clear from the context. H will be identified with A_{root}. Sequential automata will be considered a degenerate case of HAs. A central role in UMLSCs is played by *(state) configurations*, defined as follows:

Definition 9 (Configurations). *A* configuration *of HA $H = (F, E, \rho)$ is a set $\mathcal{C} \subseteq (\mathcal{S}\ H)$ such that (i) $\exists_1 s \in \sigma_{A_{root}}.\ s \in \mathcal{C}$ and (ii) $\forall s, A.\ s \in \mathcal{C} \wedge A \in \rho\ s \Rightarrow \exists_1 s' \in \sigma_A.\ s' \in \mathcal{C}$*

A *configuration* is a global state of a HA, composed of local states of component sequential automata. For $A \in F$ the set of all configurations of A is denoted by Conf_A. Moreover we will assume that for every set $\{H_1, \ldots, H_c\}$ of HAs, there exists a distinguished element $\mathcal{C}_{\mathsf{err}}$ such that $\mathcal{C}_{\mathsf{err}} \notin \bigcup_{j=1}^{c} \mathrm{Conf}_{H_j}$

Progress rule

$$\frac{\begin{array}{l} t \in \mathsf{LE}_A\ \mathcal{C}\ \beta\ ev \\ \nexists t' \in T \cup \mathsf{E}_A\ \mathcal{C}\ \beta\ ev.\ \pi t \sqsubset \pi t' \end{array}}{A \uparrow T :: \mathcal{C} \xrightarrow{(ev,\beta)/(AC\ t, \mathsf{bnd}\ ev\ (EV\ t))}_{\{t\}} DST\ t}$$

Stuttering Rule

$$\frac{\begin{array}{l} \{s\} = \mathcal{C} \cap \sigma_A \\ \rho_A\ s = \emptyset \\ \forall t \in \mathsf{LE}_A\ \mathcal{C}\ \beta\ ev.\ \exists t' \in T.\ \pi t \sqsubset \pi t' \end{array}}{A \uparrow T :: \mathcal{C} \xrightarrow{(ev,\beta)/(\epsilon,[\,])}_{\emptyset} \{s\}}$$

Composition Rule

$$\frac{\begin{array}{l} \{s\} = \mathcal{C} \cap \sigma_A \\ \rho_A\ s = \{A_1, \ldots, A_n\} \neq \emptyset \\ \left(\bigwedge_{j=1}^{n} A_j \uparrow T \cup \mathsf{LE}_A\ \mathcal{C}\ \beta\ ev :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac}_j,\xi_j)}_{L_j} \mathcal{C}_j\right) \\ \mathsf{mrg}_{j=1}^{n} \mathsf{Ac}_j\ \mathsf{Ac}\ \wedge \xi = \bigcup_{j=1}^{n} \xi_j\ \wedge L = \bigcup_{j=1}^{n} L_j \\ L = \emptyset \Rightarrow (\forall t \in \mathsf{LE}_A\ \mathcal{C}\ \beta\ ev.\ \exists t' \in T.\ \pi t \sqsubset \pi t') \end{array}}{A \uparrow T :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_{L} \{s\} \cup \bigcup_{j=1}^{n} \mathcal{C}_j}$$

Fig. 8. Rules of the Core Semantics

A.2 Core Semantics Definition

The Core Semantics definition is given in Fig. 8

As mentioned before, the Core Semantics definition is very similar to the one we have used in previous work of ours. Here we give a very brief description with emphasis on those aspects relevant for the purposes of the present paper and we refer the reader interested in more details to [11, 6]. Intuitively, $A \uparrow T :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_{L} \mathcal{C}'$ models labelled transitions of the HA A, and L is the set containing the transitions of the sequential automata of A which are selected to fire. We call $\xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_{L}$ the *STEP*-transition relation in order to avoid confusion with transitions of sequential automata. When confusion may arise, we call the latter *sequential* transitions. T is a set of sequential transitions. It represents a constraint on each of the transitions fired in the step, namely that it must not be the case that there is a transition in T with a higher priority. So, informally, $A \uparrow T :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_{L} \mathcal{C}'$ should be read as (an object the behaviour of which is specified by HA) "A, on configuration $\mathcal{C}$, provided with input event (i.e. method call) ev and the store of which is β can perform L moving to configuration $\mathcal{C}'$, when required to perform transitions with priorities not smaller than any in T; Ac is the sequence of actions to be executed as result of firing the transitions in L and ξ binds the value carried by ev, if any, to proper parameters occurring in the triggers of transitions in L". Set T will be used to record the transitions a certain automaton can do when considering its sub-automata. More specifically, for sequential automaton A, T will accumulate all transitions which are enabled in the ancestors of A. The Core Semantics definition makes use of the auxiliary functions defined in Fig. 9. $\mathsf{LE}_A\ \mathcal{C}\ \beta\ ev$

is the set of all the *enabled local* transitions of A in $\mathcal{C}, \beta$, with ev[7]. Similarly, the set of all *enabled* transitions of A—considered as an HA, i.e. including the transitions of descendants of A—in $\mathcal{C}, \beta$, with ev, is $\mathsf{E}_A\ \mathcal{C}\ \beta\ ev$.

$$\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{LE}_A\ \mathcal{C}\ \beta\ ev\ \triangleq} \\
\quad \{t \in \delta_A \mid & \{(SRC\ t)\} \cup (SR\ t) \subseteq \mathcal{C}, \\
& \mathsf{match}\ ev\ (EV\ t), \\
& (\mathcal{C}, \beta \lhd (\mathsf{bnd}\ ev\ (EV\ t)), ev) \models (G\ t)\} \\
\quad \textbf{for all} & \textit{HAs}\ H = (F, E, \rho),\ A \in F,\ \mathcal{C} \in \mathrm{Conf}_H, \\
& \textit{stores}\ \beta,\ \textit{input events}\ ev
\end{array}$$

$$\begin{array}{l}
\mathsf{E}_A\ \mathcal{C}\ \beta\ ev\ \triangleq \\
\quad \bigcup_{A' \in (\mathcal{A}\ A)} \mathsf{LE}_{A'}\ \mathcal{C}\ \beta\ ev
\end{array}$$

where:

$$\begin{array}{ll}
\mathsf{bnd}\ m\ m' & \triangleq \\
\quad [\,] & \\
\mathsf{bnd}\ m(n)\ m'(x) & \triangleq \\
\multicolumn{2}{l}{\quad \textbf{if}\ \mathsf{match}\ m(n)\ m'(x)\ \textbf{then}\ [x \mapsto n]\ \textbf{else}\ [\,]} \\
\multicolumn{2}{l}{\quad \textbf{for all}\ \ n \in \mathcal{Z}_L \cup \mathcal{Z}_O, m, m' \in \mathcal{Z}_M, x \in \mathsf{Par}}
\end{array}$$

$$\begin{array}{ll}
\mathsf{match}\ m\ m' & \triangleq \\
\quad (m = m') & \\
\mathsf{match}\ m(n)\ m'(x) & \triangleq \\
\multicolumn{2}{l}{\quad (m = m'), \mathrm{Type}[\![n]\!]_H = \mathrm{Type}[\![x]\!]_H} \\
\multicolumn{2}{l}{\quad \textbf{for all}\ \ n \in \mathcal{Z}_L \cup \mathcal{Z}_O, m, m' \in \mathcal{Z}_M, x \in \mathsf{Par}}
\end{array}$$

$$\mathrm{Type}[\![exp]\!]_H \quad \triangleq \quad \begin{array}{l} \textit{the type of expression exp in the context} \\ \textit{of (the variables/constants declaration} \\ \textit{of the class associated to) HA } H. \end{array}$$

Fig. 9. Auxiliary functions for the Core Semantics

In the Core Semantics, the Progress Rule establishes that if there is a transition t of A enabled by event ev in the current configuration $\mathcal{C}$ and store β and the priority of such a transition is "high enough" then the transition fires and a new configuration is reached accordingly. The action to be (eventually) executed is $AC\ t$ and the parameter binding is generated in the obvious way by means of function bnd. The Composition Rule stipulates how automaton A delegates the execution of transitions to its sub-automata (3rd premise) and these transitions are propagated upward. Notice that for all v, i, j, $\xi_i\ v \neq \mathsf{unbound}$ and

[7] $(\mathcal{C}, \beta, ev) \models g$ means that guard g is true for configuration $\mathcal{C}$, store β and input event ev. Its formalisation is immaterial for the purposes of the present paper. The definition of $\mathrm{Type}[\![e]\!]_H$ is part of UML static semantics and here we assume it given.

$\xi_j\ v \neq \mathsf{unbound}$ implies $\xi_i\ v = \xi_j\ v = ev$. Moreover, different orderings of actions due to different interleavings of the firing of the transitions in L are captured by means of predicate mrg (4th premise). Finally, if there is no transition of A enabled with "high enough" priority and moreover no sub-automata exist to which the execution of transitions can be delegated, then A has to "stutter", as enforced by the Stuttering Rule. Notice that stuttering of sub-automata is propagated upwards by the Composition Rule *only* if no local transition can be fired either (last premise of Composition Rule). In the operational semantics definition of Fig. 6, the simplified notation $H :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_L \mathcal{C}'$ has been used which stands for $H \uparrow \emptyset :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_L \mathcal{C}'$.

The following theorem links our semantics to the general requirements set by the official semantics of UML:

Theorem 1. *Given HA $H = (F, E, \rho)$ for all $A \in F, ev \in E, T, L, \mathcal{C}, \beta, \mathsf{Ac}$ the following holds: $A \uparrow T :: \mathcal{C} \xrightarrow{(ev,\beta)/(\mathsf{Ac},\xi)}_L \mathcal{C}'$ for some $\mathcal{C}', \xi$ iff L is a maximal set, under set inclusion, which satisfies all the following properties: (i) L is conflict-free, i.e. $\forall t, t' \in L.\ \neg t \# t'$; (ii) all transitions in L are enabled, i.e. $L \subseteq \mathsf{E}_A\ \mathcal{C}\ \beta\ ev$; (iii) there is no transition outside L which is enabled and which has higher priority than a transition in L, i.e. $\forall t \in L.\ \nexists t' \in \mathsf{E}_A\ \mathcal{C}\ \beta\ ev.\ \pi t \sqsubset \pi t'$; and (iv) all transitions in L respect T, i.e. $\forall t \in L.\ \nexists t' \in T.\ \pi t \sqsubset \pi t'$.*

Proof. The proof can be carried out in a similar way as for the main theorem of [6], by structural induction for the direct implication and by derivation induction for the reverse implication. □

Communities: Concept-Based Querying for Mobile Services

Chara Skouteli[1], Christoforos Panayiotou[1], George Samaras[1], and Evaggelia Pitoura[2]

[1] Department of Computer Science, University of Cyprus,
CY-1678 Nicosia, Cyprus
{chara, cs95gp1, cssamara}@cs.ucy.ac.cy
[2] Department of Computer Science, University of Ioannina,
GR 45110, Ioannina, Greece
pitoura@cs.uoi.gr

Abstract. In this paper, we consider semantic service discovery in a global computing environment. We propose creating a dynamic overlay network by grouping together semantically related services. Each such group is termed a community. Communities are organized in a global taxonomy whose nodes are related contextually. The taxonomy can be seen as an expandable, flexible and distributed semantic index over the system, which aims at improving service discovery. We present a distributed service discovery mechanism that utilizes communities for context-based service discovery. To demonstrate the viability of our approach, we have implemented an infrastructure for supporting communities as well as a prototype application that utilizes communities.

1 Introduction

Nowadays, a significant amount of data is stored on a variety of small devices, such as smart phones, palmtops and personal computers. These small devices are interconnected, thus composing a global network that is characterized by (i) device heterogeneity, (ii) large-scale data distribution, (iii) data heterogeneity, (iv) device mobility, and (v) a variety of communication protocols. Data stored on these small diverse devices creates what we call a global or universal database. Our goal in the DBGlobe project is to provide both the theoretical foundations and the system infrastructure for effectively querying this database [21].

To overcome differences in the communication protocols used by mobile devices and data and device heterogeneity, we employ a service oriented approach in that data are wrapped in services [14]. In this paper, we focus on the fundamental issue of how to efficiently query for services in such a global database. Service discovery in this dynamic environment, where providers and requestors are mobile, is more exigent than in the classic mobile environment where only the requestors can change location. Furthermore, the huge number of available mobile services demands an efficient service discovery mechanism.

We propose creating a dynamic overlay network above the core system to group together semantically related services, effectively creating a network of communities.

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 59–76, 2004.

Each of these communities is a set of pointers to semantically or contextually related services that are distributed over the global mobile environment (for example, a community of weather services, or a community of services provided by PDAs). Communities are distributed and are effectively organized in a global taxonomy whose nodes are related contextually. This taxonomy can be seen as an expandable, flexible and distributed semantic index over the core system, which aims at decreasing the cost of service discovery. Providing flexible service discovery over communities allows us to expand the notion of context beyond the usual concept of location. In our work, a user's context is a set of mobile services belonging to a number of different communities. Having the communities managing concept-related services provides for a more efficient service discovery.

In a nutshell, in this paper, we propose: (i) a semantic grouping of mobile services over the global computational net, effectively creating a network of communities, and (ii) a distributed service discovery mechanism that utilizes these communities for context-based service discovery. We also study two types of context-based queries, containment and continuous queries that are central in this context. To demonstrate the viability of our approach, we have implemented the infrastructure for supporting communities as well as a prototype application that utilizes this infrastructure.

The remainder of this paper is organized as follows. Section 2 gives an overview of the core system architecture, while Section 3 describes mobile service directory in terms of communities and presents the types of queries we support. Section 4 describes the taxonomy and community architecture as an overlay network. Section 5 provides examples of query execution using communities. Section 6 presents our prototype implementation. Section 7 discusses related work and finally, Section 8 presents conclusions and future work.

2 Core System Architecture

DBGlobe is a global data and service management system [21]. It connects a number of autonomous devices and provides support for describing, indexing and querying their data and services (Fig. 1). DBGlobe employs a service-oriented approach in that data are wrapped as services. The mobile devices at the perimeter of the architecture are called Primary Mobile Objects (PMOs). They may function as service providers (servers), service requestors (clients) or both. They connect to the DBGlobe system and possibly directly to each other to exchange data through services. They register by providing appropriate metadata information depending on their role. Their number and location may change over time, as new PMOs enter or leave the system and existing PMOs relocate.

Besides these "walking" miniature databases of PMOs, DBGlobe system components, dispersed throughout the stationary network, store metadata information about PMOs, users and services, provide index and directory information, and query processing capabilities. These components are called Cell Administration Server (CAS). CASs also provide low-level functionality, such as network connectivity and mobility support.

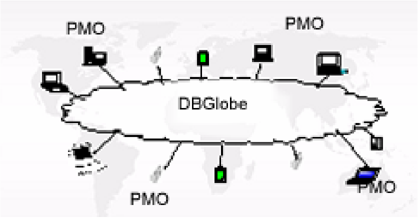


Fig. 1. The DBGlobe Layer

2.1 Primary Mobile Objects (PMOs)

A Primary Mobile Object (PMO) is any autonomous, electronic device capable of communicating independently with the CAS via some communication channel. The basic functionality of a PMO includes the ability to (a) request and retrieve data, (b) produce and share data, (c) create and publish a service and (d) communicate with a DBGlobe server and function as a source. We assume that every PMO has built-in a globally unique identity (like Ethernet adapter addresses or IMEIs in GSM phones) and possibly incorporates components that can capture context (such as GPS receivers, digital compasses and temperature sensors). In addition, it may host an application server (e.g, a web server) for executing services.

2.2 Cell Administration Servers (CAS)

The Cell Administration Servers (CASs) provide the basic DBGlobe functionality, including: (a) connectivity and addressing scheme, (b) service publication, (c) context determination support, (d) mobility support, (e) service life cycle tracking, and (f) service discovery.

We adopt a hybrid (partially ad-hoc) architecture where geographical 2-D space is divided into adjacent administrative areas (similar to GSM cells) each managed by a Cell Administration Server (Fig. 2). A network of CASs constitutes the backbone that makes it possible for the PMOs to communicate and share data and services with each other. The CASs are interconnected through a network, e.g. the Internet. Although they can function autonomously, they are also aware of their neighbors (that manage geographically adjacent cells) and cooperate to increase the range of requests. In our current design [3], each cell represents the area of coverage of s network access point. We assume that every PMO (including stationary devices) is associated with at most one cell at any given time (e.g., by keeping a live connection to the cell's defining network access point).

Each CAS can independently manage the PMOs which enter its area of authority. It keeps track of the PMOs that enter or leave the cell's boundaries. It stores metadata describing each PMO, the context and the resources offered and assists the user to locate services by semantically matching requests with existing service descriptions. It also provides basic services to visiting PMOs such as network addressing, session management and positioning. Each cell can support large numbers of PMOs moving inside its area and acting as sources or requestors of information. The CAS module consists of:

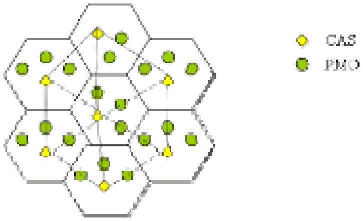


Fig. 2. The System Geographical Distribution CAS manage the covered PMOs

1. A service directory that lists all the services offered by PMOs in the cell.
2. A service description repository of the local services
3. A CAS directory, containing addresses of other CASs.
4. A community directory, containing addresses of the available communities
5. A location management sub-system that keep tracks of the location of the registered to that cell PMOs
6. A device type and a PMO repository containing the list of device types and PMOs available in the cell and their profiles,
7. A temporal profile manager for storing the connection times of devices, discovering patterns and estimating probabilities of next appearance. A server can also keep historical data and compute statistics about their mobility habits to assist proactive behavior.
8. A service discovery mechanism for locally residing services.

The distributed nature of the system, however, requires an efficient distributed service discovery mechanism which is maintained collaboratively among the various CASs. The basic idea is to use a global service director that utilizes a distributed hierarchical service taxonomy structure to assist the user to locate services by semantically matching requests with existing service descriptions. This is achieved using service communities described next.

3 Service Discovery Based on Concepts in Mobile Environment

Most common service discovery approaches are based on service registries that match service requests with available service descriptions [19]. However, in a global computing environment, where (i) service providers and requestors are mobile and (ii) service unavailability (due, for example, to wireless disconnections) occurs frequently, there is a need for more sophisticated service discovery mechanisms. For instance, users may want the results of service discovery to be adapted to their current state and be updated in a continuous fashion, e.g. if a user is driving, she is not able to type, but still wishes to find with minimum efforts results which are useful to her at the current time. To achieve this, we need an infrastructure that can collect and manage context information about the various system entities.

Context information is defined as: any information that can be used to characterize the situation of an entity, where an entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and the application themselves [17]. In the mobile environment, the most common context information includes location, user preferences, user situation and device characteristics. This information can be used to filter the results and provide more accurate and useful services. However, context information alone is not able to minimize the searching domain; in the case of a distributed system such as DBGlobe, we need to query and apply the user's context on all available service directories in each CAS. To avoid this, we use the notions of concept and communities, which allow us to contain any query within a concept efficiently and in a distributed manner.

Communities group semantically related services that are distributed over the network. Which services belong to a particular community (i.e., which services are semantically similar) is built around the notion of a concept. Concept is a semantic notion and describes a specific property, for example, "traveling", "weather" or "taxi reservation". Each concept is described through a set of keywords. Then, given a set of appropriate keywords, the matching concept or concepts (and thus communities), are identified and the appropriate services are selected. Having the universe of services divided and grouped into concepts allows a distributed and efficient implementation. Thus, if we wish to find a "weather" service for Cyprus, we just forward the query to the "weather" community and filter the results by using the location context.

3.1 Context Aware Queries

In a mobile environment, where users are moving, it is critical to provide context aware queries that (a) give concept information in a contained fashion and (b) provide results in a continuous fashion. Contained queries aim at containing the results usually within location boundaries (e.g., location-based queries). In our environment, however, the results are contained around a more general context not just location. Continuous queries are always active and aim to inform the user whenever the conditions posted in a query are satisfied (e.g. "Find services which provide videos from Greece and alert me when a new one appears").

Definition 1: A Concept Containment Query is a query that contains the results within the boundaries of a specific contextual concept. The services that are queried are related to a specific concept. Concept Containment Queries (Qccq) are composed from concept keywords and context pairs:

Qccq = <Concept{keywords}, Context{(attribute, value)}>

Concept keywords are used to identify the concept and (attribute, value) pairs are used to define the user context.

As an example, consider the case of a service for sports news. The concept in this example can be sports; the execution of the query "find me all services which provide sports news" should return all services which are related with the "sports news" concept, or a concept that is characterized by both these keywords. Depending on the query, the size of the result can be very large. Our goal is to contain the results by using the context information that characterizes the current environment of the user. For instance, in this example, if the user carries an iPAQ device, the discovered services should be suitable for it, that is, an appropriate query can be "find me all services which provide sport news displayable on an iPAQ PDA". This is expressed through the following query: Qccq = <Concept{sports, news}, Context{(device, iPAQ)}>.

In addition to containment queries, we are also interested in keeping the result of a query updated as the result set might change due to service and user mobility. This is essential since services are mobile and dynamic as PMOs may move, join or leave the system. Continuous queries aim at keeping the results of a query up-to-date. The ability to satisfy concept containment continuous queries in a global mobile environment is critical.

Definition 2: Concept containment continuous queries (Qcccq) are containment queries that notify the user for changes in the result set in a continuous fashion.

Qcccq=< Concept{keyword }, Frequency{types}, Context{(attribute, value)}>

Frequency types define the frequency by which the user should be alerted. Currently we support the "onFound" and "near by" type. An example concept containment continuous query is: "Find close by services that provide photos of Greece displayable on an iPAQ and alert me whenever a new one becomes available". This is expressed as <Concept{Greece, photos}, Frequency{onFound}, Context{(location, "current location"), (device, iPAQ)}>. This query alerts the user whenever a service which provides photos displayable on an iPAQ is near by.

Assuming that concepts are organized in some order, the order of keywords can direct query processing. Keywords following a concept ordering may improve service discovery. For example, the query "find me all services which provide sports news displayable on an iPAQ PDA" should return all services which are related with the "sports news" concept where the concept "sports news " is a sub-concept of sports. An order-preserving query language could indicate this as follows: sports | news. Other constructs such as "or", "and" could also be defined. Thus, the query above would be expressed as: <Concept{sports | news}, Context{(device, iPAQ)}>.

An important issue is how to distribute the service directories so that both containment and continuous queries are efficiently supported. CASs provide a level of distribution for the service directories, since each CAS maintains a local directory with the description of the services provided by the mobile devices (PMOs) in its coverage. By doing so, we are able to efficiently support location-based queries, that is, queries with a single location attribute. However, it is not possible to efficiently support more general concept queries, since a matching service may be registered at any CAS directory and thus all of them need to be queried. To avoid this overhead, we need a mechanism that will group services in multiple ways not only based on their location. To this end, we propose a query mechanism based on virtual directories, called communities, that cluster similar services. Communities are interconnected, creating a semantic overlay network that can be used for efficient service discovery.

3.2 Organizing Communities Using Taxonomies

Just having a collection of communities does not entirely solve the problem. We also need to organize the communities. To do so, we need a way to classify and inter-relate communities. We assume that communities are described through ontologies. Ontologies are used to fully describe entities [6, 7, 8, 10, 20], in our case, communities and services. In order to express such classifications and interrelations, we use taxonomies whose elements are the aforementioned ontologies.

Table 1. Community Ontology Properties

communityName	The name of the community
textDescription	A brief description summarizing the concept of the community
keywordsDescription	Keywords used to describe semantically the community
Parent	Reference to the parent community
Children	Reference to the children communities

Such taxonomies take the form of a tree (Fig. 3). Each internal node of the tree corresponds to an ontology that describes a community (Table 1). The node also refers to its children which are under its contextual umbrella as well as its parent node. Recursively this leads to a hierarchy of ontologies where each (deeper) level of the hierarchy provides a more refined and focused description of the concept. Each leaf of the taxonomy tree contains a subset of the description properties and functional attributes of the service's profile [20] that belongs to the (parent) community. This profile summary can be used to determine whether the service satisfies the query criteria and also to provide information for accessing the actual service (Table 2).

Given a taxonomy (i.e. a tree structure of concepts) we can run an inquiry for a specific topic by performing a top down search for matching ontologies within the taxonomy tree. During search, the parent ontologies are used to narrow down the contextual domain of the children nodes. In this way, we only get related services. For instance, we avoid asking for Asian culture services and getting hotel reservation services; we always get services from the right concept.

Using communities allows queries to run faster. The efficiency comes from the fact that communities are a collection of pointers to services belonging to a particular concept which may reside anywhere in the network. The question here is why not using one centralized unified index instead of communities. The first reason is scalability. The other one stems from the heterogeneity of the environment. A third factor is the semantic nature of the required index. We need an index that can give as the location of a service based on its semantic description, thus complicating the structure of the index. Finally, as the complexity of the index increases the cost of updating it becomes prohibiting. Communities, which are in essence a semantic index, tackle all these problems:

Table 2. Summary Service Profile Properties

serviceName	The name of the service
keywordsDescription	A keywords description summarizing semantically what service offers or what capabilities are being requested.
providedBy	A sub-property of role referring to the service provider
geographicRadius	Geographical scope of the service, either at the global scale (e.g. e-commerce) or at a regional scale (e.g. pizza delivery)
Pointer	An abstract link to the full service ontology.

- Heterogeneity in the environment does not affect communities as long as we use a standardized way of describing the available resources/data.
- Communities provide semantic based indexing of services.
- Updating a community (which has much fewer members than a unified index) is more cost effective. The updates are distributed to a number of communities, a fact that limits the load on each individual community.
- Communities can relocate as needed, thus providing load balancing.
- The CASs infrastructure provides a local index. That is, it is able to efficiently support context aware queries based on location, because the notion of location restricts the number of CASs that we have to contact.
- The ability to distribute communities provides scalability.

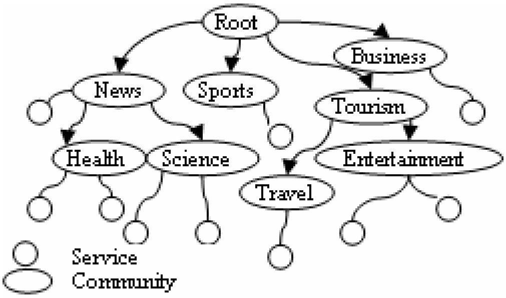


Fig. 3. Global Taxonomy of Communities and Services

Organizing the service directory around concepts and communities allows us to efficiently distribute the directory (i.e., distribute the communities) over the network.

4 Communities as an Overlay Network Over CAS

To implement communities, we introduce the notion of a Community Administrator Server (CoAS). CoASs are responsible for the creation and management of communities. Each CoAS maintains a community, which groups similar services provided by different CASs, and it can be located anywhere in the system. As the CoASs represent all communities, the complete taxonomy of the CoASs can be seen as an overlay network over the core system of CASs (Fig. 4). This overlay network instead of grouping services located in the same geographical domain, it groups services which are semantically related independently of their location. To create the overlay network of CoAs, each CAS propagates a summary of description ontologies of the services that it hosts (see Table 2) to the appropriate CoASs. Identifying the appropriate CoASs is achieved by using routing indexes based on Bloom filters [4] described in Section 4.2. The complete overlay network of CoASs constitutes a global distributed taxonomy tree of communities. Figure 4 shows a possible configuration and distribution of such a network.

4.1 Managing the CoAS Taxonomy Tree

The CoAS topology is a hierarchy of ontologies. For managing this distributed topology, important operations include (i) updating its content when a new service is registered to the system or a PMO disconnects and thus its services become unavailable and (ii) load-balancing the taxonomy when a node/community becomes overloaded with service descriptions.

Construction: As an initial global taxonomy tree, we use a basic classification of services taken from Google (e.g. a subset of Google's classification of urls). Using this classification we create the initial network of CoASs.

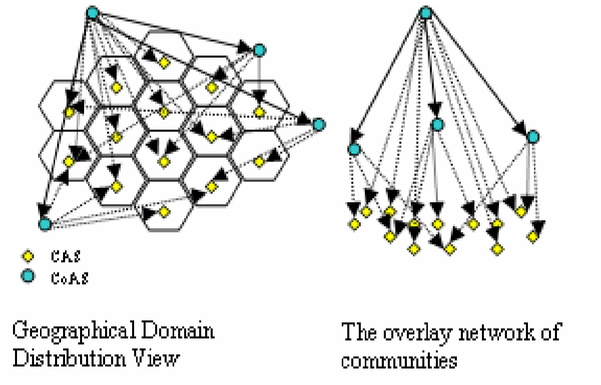

◆ CAS
● CoAS
Geographical Domain Distribution View
The overlay network of communities

Fig. 4. Distribution of CASs and CoASs

New Service Registration: This operation takes place when a PMO registers its services to the CAS. The CAS stores locally the service ontology provided by the user, and propagates the service description to the communities (it could be more than one) which share the same concept with the service. Utilizing Bloom filters [4, 5] in combination with the services' descriptions and the community concepts allow the CAS to identify the appropriate CoASs. Deletion of a service is handled in a similar manner.

Service Unavailability: A service provided by a PMO may become unavailable at any given time either voluntarily by its owner or because the PMO becomes unreachable due, for example, to network disconnections. In such cases, we do not delete the service, so during service discovery we check for the actual service availability. The CAS is responsible to detect unavailability or availability and inform the appropriate CoAS.

Service Update: An update operation at the community level takes place only when the semantic description of the service changes. In such cases, when the service profile changes, the CAS will propagate the changes only to the communities which store a summary of the service and only if this summary must be updated. Note that service mobility does not affect the community taxonomy. This is because location based queries are handled by the CASs, thus we do not have to update the communities whenever the PMO that owns the service changes location.

Balancing Communities: In case where a community becomes too large, reducing its efficiency, it can be split into two sub-communities. To decide which concept should be used to build the new community, we cluster the existing community and select the concept of the largest cluster. Two new CoASs are created to manage the new sub-communities.

Service Discovery: Querying for a service takes place when a CAS forwards a query to the CoAS that manages the community that serves the concept of the query. The CoAS is responsible to find all services which satisfy the contextual condition posted with the query. In Section 5, we present the query mechanism in more detail.

4.2 Using Bloom Filters to Locate a Community

To identify which CoAS match a given query, we use indexes based on Bloom filters. Bloom filters are compact data structures for probabilistic representation of a set that supports membership queries, that is queries on whether a given element belongs to a set. A Bloom filter BF of size m is a vector of m bits. Initially, all m bits are set to 0. Consider a set A = {a1, a2, …, an} of n elements. A number of k independent hash functions, h1, h2, …, hk, each with range 1 to m are used as follows. For each element a ∈ A, the bits at positions h1(a), h2(a), ..., hk(a) of BF are set to 1. Note that a particular bit may be set to 1 many times. Given a query for an element b, we check the bits at positions h1(b), h2(b), ..., hk(b). If any of them is 0, then certainly b is not in the set A. Otherwise we conjecture that b is in the set, although there is a probability that this is not the case. This is called a false positive. Parameters k and m can be chosen so that the probability of a false positive is acceptable.

Bloom filters are used to determine which CoAS should be updated when a new service is added, deleted or becomes unavailable from the system. They are also used to find which CoASs are relevant to a given query. In particular, given a service description, using Bloom filters, we can efficiently locate the appropriate CoASs.

At each CAS, there is one Bloom filter for each CoAS; we call this filter a community Bloom filter. Let CBF(A) be the community Bloom filter that corresponds to CoAS A. To construct the CBF(A), the k hash functions are applied to all concepts (keywords) that describe community A, and the associated bits of the filter are set to 1. Given a service s, to find the CoASs that match the service s, we apply the hash functions to each of the keywords that describe the service. For each such keyword of the service, we apply the hash functions and check which community Bloom filters match it. A filter matches the service, if all bits at the corresponding positions are set to 1. The communities that match the service s are the communities whose community Bloom Filters match all keywords describing the service.

In case that there is order among the keywords that follows the ontology schemas, we may use a query language that takes advantage of this order. This should most likely be a query language based on XPath [3] that allows us to exploit the structure of the schemas as well as their content. To this end, we have introduced multi-level Bloom filters [5] that extend Bloom filters for supporting the efficient evaluation of path expressions including partial match and containment queries. Multi-level Bloom filters are used to represent the CoAS taxonomy. In particular, instead of maintaining a simple Bloom filter for each CoAS, we maintain a multi-level one.

5 Servicing a Query Via the CoAS Network

In this section, we present how the system supports concept and continuous containment queries.

5.1 Concept Containment Queries

The query execution steps are performed in collaboration between the CAS and the CoAS components.

1. A PMO, service or user compose a query by providing the concept keywords and submit the query to the associated CAS. The order of the keywords corresponds to the concept hierarchy. The CAS composes the query by appending the context keywords which define the user current environment. As an example, consider the following request: “Find a service providing pop music clips for an iPAQ media player”. This request is formulated as follows:

   Qcq=<Concept{music, pop, clip}, Context{(device, iPAQ)}>

2. If the receiving CAS can satisfy the query then it returns the results to the issuing PMO, service or user and the process terminates (location-based queries might be satisfied this way). If the request can not be satisfied locally by the CAS, the CAS

uses the Bloom Filters to identify which community (i.e., CoAS) serves the query concept, in this example, the community “music clips”. We assume that the community taxonomy contains such a community. In this case the concept hierarchy is music | pop | clips; this hierarchy is used to better direct the query to the appropriate community. In case that there is no community to serve the exact concept, we search for a community that serves the more general concept, in our example “music pop” and the keyword “clips” is submitted as a context constraint in the query. Upon finding the appropriate CoAS, the CAS forwards the query to it. If there is no community to serve the concept, the request is forwarded to the root community for a top down search.

3. A CoAS upon receiving a query identifies all matching services. Matching is performed at a semantics level. All matching services are reported in a list. For the example query, the resulted list will contain all services which provide pop music clips currently registered in the CoAS unless other constraints are also imposed.

Example Query 1: *“Find all services providing photographs of Parthenon”*. Assuming that there is no community to serve the concept “Photographs of Parthenon” the keyword Parthenon becomes a context keyword and is used to filter the services which are registered to the photograph community. To this end, we use a reserved attribute name, called “concept”. This query is formulated as follows:
Q1 = <Concept{photograph}, Context(concept, Parthenon)}>

5.2 Concept Containment Continuous Queries

These types of queries differ from the previous ones in that they must be stored into the CoAS. Depending on the frequency condition, the CoAS will periodically push the results to the issuing PMO. To better understand this mechanism, consider the following request: “Give me all services providing music clips for an iPAQ device and alert me when a new one is available”. This request is formulated as follows:

<Concept{music, clip}, Frequency{new, onFound}, Context{(device, iPAQ)}>.

The PMO submits the query to its current CAS. The CAS forwards the query to all appropriate CoASs, using the mechanism described earlier.

Each CoAS registers the query locally. Whenever a new service is registered to the CoAS, the CoAS checks whether there is any continuous query whose conditions may be satisfied by the new service. If this is the case, the query results are updated and the issuing PMO is notified.

There is an overhead for supporting concept containment continuous queries, since we need to check whether a new service match any continuous queries registered at the corresponding CoAS. However, this overhead is small considering the overhead to provide this capability in the absence of the CoASs. In this case, all CAS would have to be checked whenever a new service is registered.

Example Query 2: *“Give me services providing photographs of Parthenon and alert me when a new one is available”*. This query is expressed as follows:

Q2 = <Concept{photograph}, Frequency{new, onFound}, Context{(concept, Parthenon}>

Example Query 3: *"Give me services providing finance services and alert me when a new one is submitted by the user "xak"*. This query must be forwarded to community managing the concept "Finance" and with context the service provider. This is expressed as:

Q3 = <Concept{finance}, Frequency{new, onFound}, Context{(provider, "xak")}>

6 Implementation and Prototype

The core system infrastructure is composed by a set of CASs. These CASs are distributed across the network and independently manage the PMOs under their area of coverage. Low level communication between the available CASs is achieved by using RMI. For extensibility, we also manipulate CASs as web services. The CAS interface includes methods for (i) registering a new service, (ii) locating a service and (iii) retrieving context information (e.g. location).

We implemented the Community Administrator Servers (CoASs) on top of the core system infrastructure. CoASs are also distributed across the network and can be manipulated as web services. The main system components which have direct access to the CoASs are the CASs. Communication of these components is achieved either with RMI or web services technology. The interface provide by a CoAS consists of the following methods: (i) register a new service, (ii) locate a service, and (iii) get the results of a continuous query.

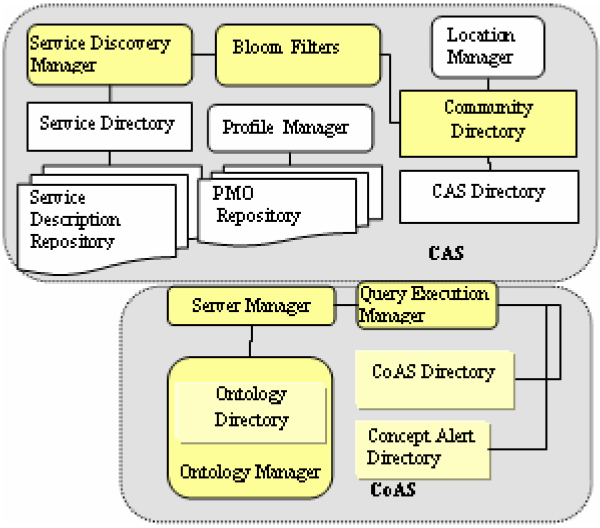
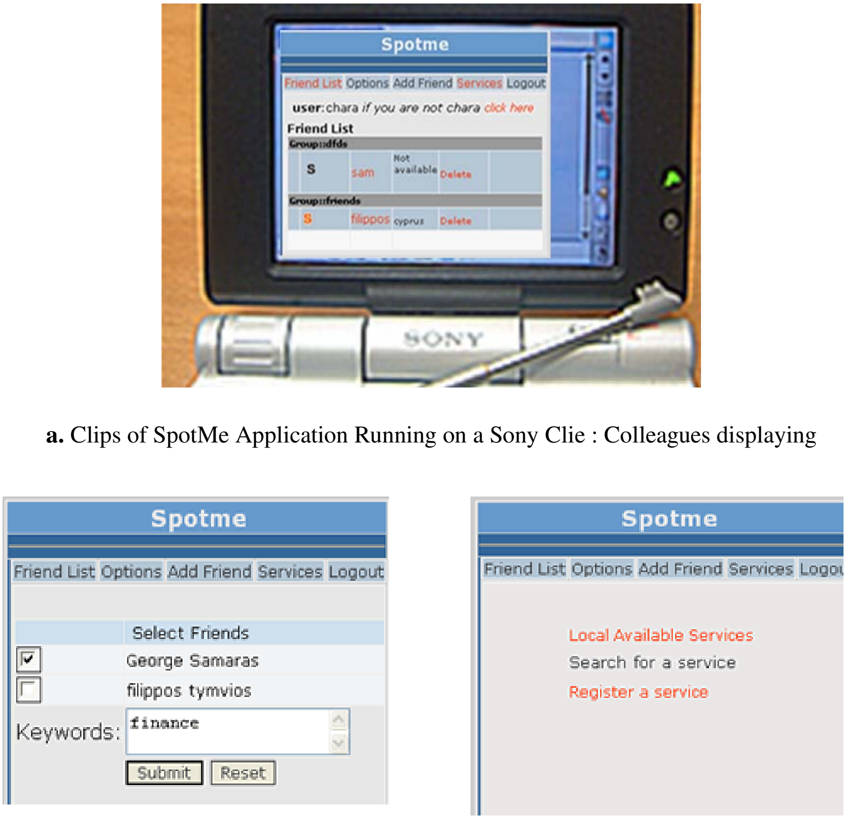


Fig. 5. The CAS and CoAS Server Architecture

6.1 Community Administrator Server (CoAS) Architecture

The components that comprise a CoAS are the following (Fig. 5):
Service Ontology Directory: lists all the service ontologies summaries currently handled by the specific CoAS.

1. CoAS directory: lists children CoAS.
2. Query executor: the most important component of the CoAS, as it is responsible for matching an incoming query's criteria with service describing ontologies. In effect, it is the context awareness query processor.
3. Concept Alerts Directory: used to better support continuity for incoming queries by providing triggers for them.
4. CAS Directory: lists all the CAS of the system.



a. Clips of SpotMe Application Running on a Sony Clie : Colleagues displaying

b. Creation of a Concept Containment Continuous Query

c. View the Results of the Alert

Fig. 6. SpotMe Prototype

6.2 SpotMe: A Context Aware Application

One of the main objectives of our infrastructure is to support the development of context aware applications. To demonstrate the capabilities of our system, especially distributed service discovery, we implemented a context aware prototype application called SpotMe on top of the CAS and CoAS infrastructure. The goal of the application is to create a collaborative environment where groups of users connect to the system to share their services. The prototype application is web-based and supports both continuous and containment queries.

Figure 6 exhibits some of the application capabilities. Fig. 6.a shows the basic screen of the application where a user has organized her friends into groups. As shown in Fig. 6.c, the user is able to view the local available services; this option exhibits the infrastructure capabilities to provide location-based queries. Moreover, the user has the option to register her services and search for services. Figure 6.b details the search capability where the user selects a list of friends and also submits a set of keywords which define the concept of the services. A possible concept containment continuous query example is “Inform me when one of my friends submits a new financial service” where the concept of this query is “financial” and the context is the selected list of users. To demonstrate concept containment continuous queries, the application offers the option to users to be alerted whenever one of their selected friends registers a related service. Thus, the search in Fig. 6.b is translated to the query:

<Concept{financial}, Frequency{new, onFound}, Context{(users, friends)}.

The following environment has been used to test the prototype implementation: a CAS network consisting of three CAS interconnected through the internet. All of them are also internet gateways, two of them allowing near-by users to access the network via wifi and one of them via Bluetooth. The initial overlay network of communities, shown in Fig. 7, is also interconnected through the internet. We used the following mobile devices: a Sony Clie, a Toshiba and an iPAQ connected via wifi and Bluetooth. Figure 6.a shows clips of the application’s interface on a Sony clie.

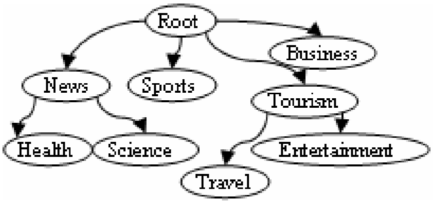


Fig. 7. Initial Community Hierarchy

7 Related Work

GloServ [12] is a service discovery system for a mobile environment that shares same common design issues with our work. More specifically, GloServ uses a hierarchical schema to classify the registered services. Its architecture is similar to DNS in that it contains root name servers and authoritative name servers that manage information about services. GloServ classifies the hierarchy of services and establishes RDF schemas that describe each type of service. The SLM system [18] is another service discovery architecture that shares some ideas with our approach. The SLM service discovery system consists of SLM servers, services and SLM clients. An SLM server is a service information repository, providing SLM clients with access to all available services. SLM clients can search for services on behalf of end users. The system adopts a distributed hierarchical tree structure to organize SLM servers which may physically be located in wide-area networks. Both approaches create a hierarchy between the directory servers where the services are registered, while, in our approach the directory servers (CASs) are interconnect in a graph structure; but we provide the hierarchy of the available services on top of this graph structure. Our approach is more scalable because when a CAS does not respond, we are still able to find a service because all services are indexed by the taxonomy tree. Conversely in the case that a CoAS is not available, the query can be executed by the CAS.

The SCAM [13] context model is based on an ontology which provides a vocabulary for representing and sharing context knowledge in a pervasive computing domain, including machine-interpretable definitions of basic concepts in the domain and relations among them. To capture a great variety of context, they divide a pervasive computing domain into several sub-domains, e.g., home domain, office domain, vehicle domain, etc; and define individual low-level ontology in each domain. The separation of domains reduces the burden of context processing and makes context interpretation possible on mobile thin clients. The important difference with our approach and SCAM is that SCAM uses a centralized architecture, OSGi-compliant, mobile service gateway.

8 Conclusions and Future Work

In this paper, we consider semantic service discovery in a global computing environment. We described a low-level architecture of directory servers, each one of which maintains information about the services offered by the devices inside its area of coverage. We proposed creating a dynamic overlay network above this network of servers that groups semantically related services, effectively creating a network of communities. Each community is a set of pointers to semantically or contextually related services (for example, a community of weather services). Communities are organized in a global taxonomy whose nodes are related contextually. This taxonomy can be seen as an expandable, flexible and distributed semantic index over the core system, which aims at improving service discovery. We also presented a distributed service discovery mechanism that utilizes these communities for context-based service discovery. To demonstrate the viability of our approach, we have implemented the infrastructure for supporting communities as well as a prototype application that utilizes this infrastructure. As future work, we plan to explore the effectiveness of a query language for managing context. We also plan to study load-balancing by relocating communities close to their most frequent requestors.

References

1. Samaras,G., Spyrou,K., Pitoura,E., Dikaiakos, M.: Tracker: A Universal Location Management System for Mobile Agents. Proc. The European Wireless 2002 Conference, Next Generation Wireless Networks: Technologies, Protocols, Services and Applications, Florence, Italy (2002) 572–580
2. Bray, T., Paoli , J., Sperberg-McQueen, C. M.: Extensible Markup Language (XML) 1.0 Specifications. World Wide Web Consortium, http://ww.w3.org/TR/Rec-xml
3. XML Path Language (XPath). World Wide Web Consortium, http://www.w3.org/TR/xpath
4. Koloniari, G., Pitoura, E.: Content-Based Routing of Path Queries in Peer-to-Peer Systems. EDBT Heraclio Greece (2004) 29–47
5. Koloniari,G., Pitoura, E.: Bloom-Filters for Hierarchical Data, Proceeding of the 5th Workshop on Distributed Data and Structures (WDAS) (2003)
6. Services Definition Language (WSDL), Web page, http://www.w3.org/TR/WSDL.
7. Ouzzani, M., Benatallah, B., Bouguettaya, A.: Ontological Approach for Information Discovery in Internet Databases. Distributed and Parallel Databases an International Journal, Volume 8, Issue 3 (2000) 367–392
8. Levy, A. , Srivastava, D., Kirk., T.: Data model and query evaluation in global information systems. Intelligent Information Systems, 5(2) (1996)
9. Lee,C., Helal, D.: Context Attributes: An Approach to Enable Context-awareness for Service Discovery. In the Proceedings of the 2003 Symposium on Applications and the Internet,(SAINT'03), Orlando, FL, USA, (2003)
10. Pfoser,D., Tryfona, N.,Verykios, V.: Services-Based Data Management in a Global Computing Environment. Workshops (WISEW'03) Roma (2003) 45-53
11. XML Query (XQuery). World Wide Web Consortium, http://www.w3.org/XML/Query
12. Arabshian ,K., Schulzrinne, H.: GloServ: Global Service Discovery Architecture, Department of Computer Science, Columbia University, New York (2004)
13. Tao Gu , Xiao Hang Wang , Hung Keng Pung , Da Qing Zhang : A Middleware for Context-Aware Mobile Services, IEEE Vehicular Technology Conference (VTC Spring 2004), Milan, Italy (2004)
14. http://www.w3.org/2002/ws/
15. http://www.w3.org/2001/04/30-tbl
16. Pitoura,E., Samaras, G., :Locating Objects in Mobile Computing. IEEE Transactions on Knowledge and Data Engineering Journal (TKDE). Vol. 13, No. 4 (2001) 571–592
17. Dey, A.K. , Abowd, G.D.: Towards a Better Understaning Context and Context-Awareness. In the Workshop on The What, Who, Where, When, and How of Context-Awareness, The Hague, The Netherlands (2000)
18. Gu,T., Qian, H. C. ,Yao, J. K., Pung, H. K. :An Architecture for Flexible Service Discovery in OCTOPUS", Proc. of the 12th International Conference on Computer Communications and Networks (ICCCN), Dallas, Texas (2003)
19. UDDI: The UDDI Technical White Paper. http://www.uddi.org

20. DAML-S Coalition:DAML-S Service Description for the Semantic Web, In Proceedings of The First International Semantic Web Conference (ISWC) Sardinia, Italia (2002)
21. Pitoura,E.,Abiteboul, S., Pfoser, D.,Samaras, G., Vazirgiannis, M., et. al. : DBGlobe: a Service-Oriented P2P System for Global Computing, SIGMOD Record 32(3) (2003) 77–82

Towards a Formal Treatment of Secrecy Against Computational Adversaries

Angelo Troina[1], Alessandro Aldini[2], and Roberto Gorrieri[3]

[1] Dipartimento di Informatica, University of Pisa,
Via F. Buonarroti 2, 56127 - Pisa, Italy
troina@di.unipi.it

[2] Istituto STI, University of Urbino,
Piazza della Repubblica 13, 61029 - Urbino, Italy
aldini@sti.uniurb.it

[3] Dipartimento di Scienze dell'Informazione, University of Bologna,
Mura Anteo Zamboni 7, 40127 - Bologna, Italy
gorrieri@cs.unibo.it

Abstract. Polynomial time adversaries based on a computational view of cryptography have additional capabilities that the classical Dolev-Yao adversary model does not include. To relate these two different models of cryptography, in this paper we enrich a formal model for cryptographic expressions, originally based on the Dolev-Yao assumptions, with computational aspects based on notions of probability and computational power. The obtained result is that if the cryptosystem is robust enough, then the two adversary models turn out to be equivalent. As an application of our approach, we show how to determine a secrecy property against the computational adversary.

1 Introduction

The recent literature concerning the analysis of security protocols reveals an increasing interest towards the compatibility problem between the computational approach, followed by the cryptography community, and the approach based on the Dolev-Yao model, which is instead followed by the formal methods community (see, e.g., [2, 5, 11, 22, 15, 14, 18, 6]). In particular, it has been widely recognized that a sort of computational view of cryptography must be introduced in the formal approaches to security analysis based on a purely formal treatment of cryptographic operations. The classical Dolev-Yao model, which is based on the perfect cryptography assumption and the restricted expressive power of the adversary [8], favours a convenient application of formal methods that treat cryptographic operations as purely formal. In this view, an encrypted message, which is a formal expression, can be suitably analyzed through techniques borrowed from the fields of, e.g., modal logic and process algebra [16, 12, 10, 19, 17, 9]. On the contrary, such a model does not take into account that the adversary has (limited) computational resources which can be exploited to obtain data in a

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 77–92, 2004.

way that cannot be captured by, e.g., standard inference rules. The adversary advantage is instead based on notions of probability and computational power.

On the basis of such considerations, we aim at relaxing the strict requirements of the Dolev-Yao approach to cryptography. In order to overcome the limitations of such requirements, we take into account the probability for a polynomial time adversary of attacking with success a message encrypted with a secret key. While in a Dolev-Yao setting such a possibility is simply disregarded – a message encrypted with an unknown key is a black box – in a real scenario an adversary with a suitable knowledge may have a good chance of obtaining useful information from a ciphertext that, from a purely formal standpoint, is considered to be a black box. By considering the probability of cryptanalyzing a ciphertext, we compare cryptographic expressions through a suitable notion of indistinguishability, which has been introduced in [21]. Such a notion, which is based on a similarity relation, states whether a polynomial time adversary with a certain initial knowledge has a non-negligible probability of distinguishing two different cryptographic expressions. As a simple example, expressions $\{M\}_K$ and $\{rubbish\}_K$ are almost the same if K is secret and the encryption scheme is *ideal* according to a computational view of what, in practice, perfect cryptography stands for (see, e.g., [2, 11]).

In this paper, we show that the definition of similarity for cryptographic expressions corresponds to the classical Dolev-Yao based notion of equivalence in the case a suitably defined encryption scheme is used that, intuitively, turns out to be robust against any cryptanalysis attack conducted by a polynomial-time adversary. In practice, the intuition is that if the cryptosystem is robust enough, then a computational adversary with a limited amount of resources has the same expressive power of an adversary that does not use cryptanalysis to obtain data. As an application, we show that our notion of similarity can be used to determine the secrecy degree of a message within an encrypted expression.

The rest of the paper is organized as follows. First, we describe how we extended the Dolev-Yao formal model with probabilistic information used to estimate the probability for a polynomial-time adversary of obtaining meaningful information from a ciphertext (Sect. 2). Then, we show a similarity relation that allows cryptographic expressions to be compared from the viewpoint of a polynomial-time adversary (Sect. 3). Afterwards, we present the main theorem showing that such a similarity relation corresponds to the equivalence relation of the Dolev-Yao model in the case the encryption scheme is robust enough (Sect. 4). As an example, we show an application of such an approach to a secrecy problem in system security analysis (Sect. 5). Finally, we conclude the paper by discussing some related work (Sect. 6) and future work (Sect. 7).

2 Equivalence and Computational Adversary

We base our formal model on the Dolev-Yao encryption model defined by Abadi and Rogaway [2]. In this setting, we formulate an extension of the classical equivalence relation among cryptographic expressions that allows for relating those expressions that yield the same information obtained with the same probability even through cryptanalysis attempts. Therefore, we abandon the usual Dolev-Yao abstraction and we take into account cryptanalysis attacks.

2.1 Setting the Context

As a preliminary to our extension, we now introduce the machinery needed to compare cryptographic expressions. We use **String** to denote a finite set of plaintext messages, i.e. the set of binary strings of a fixed length (ranged over by $m, n, \ldots$), **Keys** to denote a fixed, non-empty set of key symbols (ranged over by $K, K', K'', \ldots$ and $K_1, K_2, K_3, \ldots$), such that **Keys** and **String** are disjoint, and **Exp** to denote the set of *expressions* defined by the grammar:

$M, N ::=$	expressions
K	key (for $K \in \mathbf{Keys}$)
m	string (for $m \in \mathbf{String}$)
(M, N)	pair
$\{M\}_K$	encryption (for $K \in \mathbf{Keys}$)

Intuitively, (M, N) represents the pairing of M and N, and $\{M\}_K$ represents the encryption of M under K via a symmetric encryption algorithm. Pairing and encryption can be nested, like, e.g., in $(\{(m, K)\}_{K_1}, K_1)$.

The *entailment* relation $M \mapsto N$ says that N can be derived from M. Formally, such a relation is inductively defined as the least relation satisfying the following properties:

$$\boxed{\begin{array}{ll}
M \mapsto M & \\
M \mapsto N_1 \ \wedge \ M \mapsto N_2 & \Rightarrow M \mapsto (N_1, N_2) \\
M \mapsto (N_1, N_2) & \Rightarrow M \mapsto N_1 \ \wedge \ M \mapsto N_2 \\
M \mapsto N \ \wedge \ M \mapsto K & \Rightarrow M \mapsto \{N\}_K \\
M \mapsto \{N\}_K \ \wedge \ M \mapsto K & \Rightarrow M \mapsto N
\end{array}}$$

In essence, $M \mapsto N$ expresses what the adversary obtains from M without any prior knowledge of its content. For instance, $(\{\{K_1\}_{K_2}\}_{K_3}, K_3) \mapsto K_3$, and $(\{\{K_1\}_{K_2}\}_{K_3}, K_3) \mapsto \{K_1\}_{K_2}$, but $(\{\{K_1\}_{K_2}\}_{K_3}, K_3) \not\mapsto K_1$. The entailment relation models the expressive power of the adversary based on the Dolev-Yao model and includes all the operations that such an adversary can execute to construct ciphertexts or extract plaintexts.

Our extension consists in taking into account the possibility for an adversary of obtaining meaningful information from a ciphertext $\{M\}_K$ without knowing the key K. To this purpose, we introduce the *probabilistic pattern* $P._p$, which represents an expression P that does not contain undecryptable blocks and is associated with a parameter $p \in]0, 1]$, which models the probability of getting the plaintext contained in P. Formally, we define the set **pPat** of probabilistic patterns with the grammar:

$P_{\cdot p}, Q_{\cdot p} ::=$	probabilistic patterns
$K_{\cdot p}$	key (for $K \in$ **Keys**)
$m_{\cdot p}$	string (for $m \in$ **String**)
$(P_{\cdot p}, Q_{\cdot p})_{\cdot p}$	pair
$p \in]0, 1]$	

A probabilistic pattern associated to a ciphertext is obtained by substituting every ciphered block with the corresponding expression in clear associated with the probability of obtaining information about it. Given a computational polynomial time adversary $\mathcal{A}$ and an initial knowledge G, the probabilistic pattern associated with expression $\{m\}_K$ is expressed in terms of the probability of obtaining information about m, denoted by $m.p_{dec}(\{m\}_K, G)$. Function $p_{dec}(\{m\}_K, G)$ returns the probability of obtaining meaningful information from the ciphertext $\{m\}_K$ by exploiting the initial knowledge G. More formally, an adversary $\mathcal{A}$ with polynomially timed resources and knowledge G has a probability Pr at most equal to the value expressed by p_{dec} of retrieving K from $\{m\}_K$:

$$\boxed{Pr[K \leftarrow \mathcal{A}(\{m\}_K, G)] \leq p_{dec}(\{m\}_K, G) \text{ for all } \mathcal{A}}$$

Note that the outcome of p_{dec} is a value strictly greater than 0, because, even if with small probability, an adversary may randomly guess the key. Besides, the value of p_{dec} depends on the knowledge G exploited to conduct the cryptanalysis attempt. Intuitively, we could figure out the adversary as an arbitrary (probabilistic) algorithm, executing in polynomial time, that makes computations on ciphered blocks in order to get information about the ciphering key and the contained plaintext (see, e.g., [11]). Note that the classical Dolev-Yao adversary obtains K from $\{m\}_K$ if and only if K can be derived from G: If $G \mapsto K$, then $p_{dec}(\{m\}_K, G) = 1$. On the other hand, in a computational model assuming ideal encryption [11] or type-0 secure encryption scheme [2], p_{dec} is a negligible function, as it turns out that the probability of guessing information that cannot be derived through the Dolev-Yao model of cryptography is negligible. In the following we will consider a formal definition of negligible function and we will show that if p_{dec} is negligible, then it holds that the expressive power of the computational adversary is limited by that of the Dolev-Yao adversary and vice versa.

The outcome of function p_{dec} represents the starting point for the computation of the probability of cracking a ciphered block. Consider, e.g., expression $(\{\{m\}_{K_1}\}_{K_2}, \{(K_1, K_2)\}_K)$. What is the probability of getting information about m in the case no prior knowledge is available? An immediate answer could be $p_{dec}(\{\{m\}_{K_1}\}_{K_2}) \cdot p_{dec}(\{m\}_{K_1})$ [1], that is the probability of sequentially crack-

[1] For the sake of simplicity, we omit the knowledge G whenever either G is equal to the empty set or the content of G is clear from the context.

ing the two keys K_2 and K_1. However, we observe that if K is a weak key, then information about K_1 and K_2 can be easily derived from $\{(K_1, K_2)\}_K$ and, as a consequence, the cryptanalysis of $\{\{m\}_{K_1}\}_{K_2}$ may be simplified. Hence, the probability of success may vary according to the strategy adopted by the adversary. By considering the worst case, we always associate to a ciphered block the maximum probability of getting information about it, i.e. we take into account the best cryptanalysis strategy from the adversary standpoint. To this end, we analyze all the possible cryptanalysis paths that the adversary can follow. In the next section, we describe through an illustrative example the structures and the functions used to determine the best cryptanalysis strategy [21].

2.2 Cryptanalyzing a Ciphertext

The methodology that aims at turning an expression N into a probabilistic pattern $N._p$ consists of four steps. In this section we illustrate each step of the methodology through an illustrative example. We consider the expression $N = (\{\{m\}_{K_1}\}_{K_2}, \{(K_1, K_2)\}_K)$ and we assume that the initial knowledge G does not allow the adversary to derive any information from N through the entailment relation.

The first step of our methodology consists in computing the keys that can be obtained from the expression, possibly with (without) cryptanalysis attempts. In our example, by hypothesis it is not possible to obtain keys through the entailment relation. In other words, the expression is a black box from the viewpoint of a classical Dolev-Yao adversary. In fact, we have that K can be derived with a probability based on $p_{dec}(\{(K_1, K_2)\}_K)$. Through such a single cryptanalysis, the adversary obtains K_1 and K_2 too. Alternatively, the adversary may try to obtain K_2 by attacking $\{\{m\}_{K_1}\}_{K_2}$ with a success probability equal to $p_{dec}(\{\{m\}_{K_1}\}_{K_2})$. Afterwards, $\{m\}_{K_1}$ may be cracked in a similar way, and so on. In essence, several different strategies can be adopted to derive the keys contained in N – K, K_1, and K_2 – from N itself, and each of such strategies must be evaluated.

Formally, given an expression M and the initial knowledge G, we denote by $\mathbf{pKeys}^G_M$ a set of pairs of the form $T._p$, where $T \subseteq \mathbf{Keys}$ is a set of keys syntactically occurring in M, and $p \in]0, 1]$ is the probability of retrieving the keys contained in T through a certain strategy. Set $\mathbf{pKeys}^G_M$ is generated through the following two-phase algorithm:

$$\begin{array}{l} \mathbf{pKeys}^G_M = \{initKeys((M, G))._1\}; \\ addKeys((M, G), 1); \end{array}$$

In the first phase, $\mathbf{pKeys}^G_M$ is initialized with the set of keys that can be derived from M and G without cryptanalysis attempts. Formally, $initKeys$: $\mathbf{Exp} \to \mathcal{P}(\mathbf{Keys})$ takes an expression L and returns the set of keys recoverable from L through the entailment relation. Hence, $initKeys(L) = \{K \in \mathbf{Keys} \mid L \mapsto K\}$. Then, in the second phase, the probabilities of retrieving the remaining keys contained in M are calculated. Formally, $addKeys(H, p)$, with $H \in \mathbf{Exp}$ and $p \in]0, 1]$, is defined through the following algorithm:

```
$addKeys(H,p)$ ::=
    $\forall \{N\}_K \ : \ (H \mapsto \{N\}_K \ \wedge \ H \not\mapsto K)$ do begin
        $p'$              $= p \cdot p_{dec}(\{N\}_K, H)$
        $L$               $= (H, K)$
        $T$               $= \{K \in \mathbf{Keys} \mid L \mapsto K\}$
        $\mathbf{pKeys}_M^G$       $= \mathbf{pKeys}_M^G \cup \{T_{\cdot p'}\}$
        $addKeys(L, p')$
    end
```

At each step of the algorithm above, a cryptanalysis is performed that reveals, with a certain probability, new keys obtained from M. In particular, for each cryptanalysis strategy, $\mathbf{pKeys}_M^G$ contains the set of keys violated by following that strategy and the probability of cracking such keys.

The second step of our methodology consists in computing the maximum probability of retrieving from an expression a given set of keys by following the best cryptanalysis strategy. In our example, the maximum probability of guessing K is equal to the probability of cracking $\{(K_1, K_2)\}_K$, because this is the only strategy that can be followed to obtain K. On the other hand, the maximum probability of guessing K_2 is the maximum between $p_{dec}(\{\{m\}_{K_1}\}_{K_2})$ and $p_{dec}(\{(K_1, K_2)\}_K)$, which are the probabilities associated with the two possible strategies that can be followed to obtain K_2.

Formally, given an expression M, the initial knowledge G, and a set T of keys included in M, we denote by $pGuess_M^G(T)$ the maximum probability of cracking all the keys in T according to the best cryptanalysis strategy that can be followed to attack M. Denoted $Keys(M)$ the set of keys occurring in M and given the set $\mathcal{D}_{pGuess_M^G} = \{T \subseteq \mathbf{Keys} \mid T \subseteq Keys(M)\}$, we define $pGuess_M^G : \mathcal{D}_{pGuess_M^G} \rightarrow]0,1]$ as:

$$pGuess_M^G(T) = \max\{p \mid J_{\cdot p} \in \mathbf{pKeys}_M^G \ \wedge \ T \subseteq J\}.$$

Note that $pGuess_M^G(\emptyset) = 1$ since $initKeys((M,G))_{\cdot 1} \in \mathbf{pKeys}_M^G$ and $\emptyset \subseteq initKeys((M,G))$.

The third step of our methodology consists in computing the maximum probability of retrieving all the information contained in a ciphertext. In our example, that means we want to evaluate the maximum probability of getting m, K, K_1, and K_2. As it is easy to see, such a probability is equal to $p_{dec}(\{(K_1, K_2)\}_K)$, because through such a single cryptanalysis it is possible to obtain all the keys used within N.

Formally, given an expression M and the initial knowledge G, we denote by $pMax_M^G$ the maximum probability of getting the information contained in M:

$$pMax_M^G = pGuess_M^G(Keys(M)).$$

The fourth step of our methodology consists in turning each ciphered block of an expression into a probabilistic pattern. The obtained probabilistic patterns associate each plaintext with the maximum probability of obtaining it. In our example, the ciphertext $\{(K_1, K_2)\}_K$ is turned into the probabilistic pattern

$(K_1 \cdot_p, K_2 \cdot_p) \cdot_p$ where $p = pGuess_M^G(\{K\}) = p_{dec}(\{(K_1, K_2)\}_K)$ and the ciphertext $\{\{m\}_{K_1}\}_{K_2}$ is turned into the probabilistic pattern $m \cdot_{pGuess_M^G(\{K_1, K_2\})}$.

Formally, given an expression M, the initial knowledge G, and an expression M' contained in M, we denote by $pP_M^G(M', T)$ a function that (i) returns in T the set of keys needed to obtain the plaintext contained in M' and (ii) associates to such a plaintext the maximum probability of obtaining it through the best cryptanalysis strategy that can be applied to M. Function $pP_M^G : \mathbf{Exp} \times \mathcal{D}_{pGuess_M^G} \rightarrow \mathbf{pPat}$ is defined inductively as follows:

$$\begin{array}{lll}
pP_M^G(K, T) & = K \cdot_{pGuess_M^G(T)} & (K \in \mathbf{Keys}) \\
pP_M^G(m, T) & = m \cdot_{pGuess_M^G(T)} & (m \in \mathbf{String}) \\
pP_M^G((N_1, N_2), T) & = (pP_M^G(N_1, T), pP_M^G(N_2, T)) \cdot_{pGuess_M^G(T)} & \\
pP_M^G(\{N\}_K, T) & = pP_M^G(N, T') & (T' = T \cup \{K\})
\end{array}$$

Finally, given an expression M and the initial knowledge G, the probabilistic pattern associated to M is given by $pP_M^G(M, \emptyset)$. In the following, we use the abbreviation pP_M^G (with no arguments) to stand for $pP_M^G(M, \emptyset)$.

The values $pMax_M^G$ and pP_M^G yield different information that are both meaningful to relate cryptographic expressions. Consider the following expressions:

$$M = (\{m\}_K, \{n\}_K) \text{ and } N = (\{m\}_K, \{n\}_{K'}), \text{ with } K \neq K'.$$

which yield the same probabilistic patterns. Indeed [2]:

$$pP_M = (m \cdot_{\hat{p}}, n \cdot_{\hat{p}}) \cdot_1,$$

where $\hat{p} = pGuess_M(\{K\}) = \max\{p_{dec}(\{m\}_K), p_{dec}(\{n\}_K)\}$. The intuition is that an adversary can crack M by guessing K, which is used to cipher both blocks. However, if $pGuess_M(\{K\}) = pGuess_N(\{K\}) = pGuess_N(\{K'\})$ we also have that:

$$pP_N = (m \cdot_{\hat{p}}, n \cdot_{\hat{p}}) \cdot_1.$$

Hence, M and N have the same probabilistic pattern, even if to get in clear the whole expression N an adversary should guess two different keys, namely K and K'. Such a difference is captured by the fact that:

$$pMax_M = pGuess_M(\{K\}) = \hat{p} \neq \hat{p}^2 = pGuess_N(\{K, K'\}) = pMax_N.$$

Therefore, $pMax_M^G$ is needed to express the overall probability of getting the whole plaintext, while pP_M^G is needed to associate each piece of information contained in an expression with the probability of getting it in clear.

In the following, we show how the information computed through the methodology surveyed above can be exploited to compare cryptographic expressions.

[2] We assume an empty knowledge and omit G.

2.3 Probabilistic Equivalence

Given the expressions M and N and an initial knowledge G, we say that M and N are *probabilistically equivalent* ($M \approx^G N$) if they yield the same probabilistic pattern and if $pMax_M^G$ and $pMax_N^G$ are equal.

Definition 1. $M \approx^G N \quad \Leftrightarrow \quad pP_M^G = pP_N^G \ \wedge \ pMax_M^G = pMax_N^G.$

Example 1. Consider the expressions $N = (\{\{m\}_{K_1}\}_{K_2}, \{(K_1, K_2)\}_K)$ and $M = (\{m\}_{K_1}, \{(K_1, K_2)\}_K)$, and assume an empty initial knowledge. If K is weaker than K_1, we have that $p_{dec}(\{m\}_{K_1}) \le p_{dec}(\{(K_1, K_2)\}_K)$ and $pGuess_M(\{K_1\}) = pGuess_M(\{K\}) = p_{dec}(\{(K_1, K_2)\}_K)$. Therefore, given $\hat{p} = p_{dec}(\{(K_1, K_2)\}_K)$, we have $pP_M = (m._{\hat{p}}, (K_1._{\hat{p}}, K_2._{\hat{p}})._{\hat{p}})._1$. On the other hand, from the previous examples and from the condition $p_{dec}(\{m\}_{K_1}) \le p_{dec}(\{(K_1, K_2)\}_K)$, we obtain the probabilistic pattern $pP_N = (m._{\hat{p}}, (K_1._{\hat{p}}, K_2._{\hat{p}})._{\hat{p}})._1$ and, since $pMax_M = pMax_N = \hat{p}$, we also obtain $M \approx N$. In conclusion, we observe that ciphering the first block m of N with both keys K_1 and K_2 is not meaningful, since N is probabilistically equivalent to an expression where this information is ciphered with one of those keys only. Indeed, an adversary can gain information about m by cryptanalyzing the second block $\{(K_1, K_2)\}_K$.

3 Indistinguishability

The notion of equivalence presented above is not adequate to state the indistinguishability among cryptographic expressions. On the one hand, it is not realistic to require that the same ciphered blocks have to be decrypted exactly with the same probability. On the other hand, it is not worth considering those blocks that can be decrypted with negligible probability, since essentially they are *almost* equivalent to a black box.

In order to relax the notion of equivalence for cryptographic expressions, we introduce a relation, called *$\varepsilon-$probabilistic similarity* ($\approx_\varepsilon$), which (i) approximates the equivalence by introducing a tolerance to small differences of the probabilistic parameters that are associated with the probabilistic patterns, and (ii) allows for equating the black box and those ciphertexts that can be decrypted with a negligible probability.

Given an initial knowledge G, we say that M and N are $\varepsilon-$probabilistically similar ($M \approx_\varepsilon^G N$) if $pMax_M^G$ and $pMax_N^G$ are *almost the same* up to the tolerance ε and if M and N are $\varepsilon-$compatible according to the notion of compatibility $\sim_\varepsilon$ specified below.

Definition 2. $M \approx_\varepsilon^G N \quad \Leftrightarrow \quad pP_M^G \sim_\varepsilon pP_N^G \ \wedge \ |pMax_M^G - pMax_N^G| \le \varepsilon.$

The compatibility relation $\sim_\varepsilon$ for probabilistic patterns expresses when two probabilistic patterns are indistinguishable. Formally, it is defined as follows:

$$\begin{array}{|lll|}
\hline
P_{\cdot p} \sim_\varepsilon Q_{\cdot p'} & \mathit{if}\ p, p' \leq \varepsilon & P_{\cdot p}, Q_{\cdot p'} \in \mathbf{pPat} \\
K_{\cdot p} \sim_\varepsilon K_{\cdot p'} & \mathit{if}\ |p - p'| \leq \varepsilon & K \in \mathbf{Keys} \\
m_{\cdot p} \sim_\varepsilon m_{\cdot p'} & \mathit{if}\ |p - p'| \leq \varepsilon & m \in \mathbf{String} \\
(P_{\cdot p_1}, Q_{\cdot p_2})_{\cdot p_3} \sim_\varepsilon (P'_{\cdot p'_1}, Q'_{\cdot p'_2})_{\cdot p'_3} & \mathit{if}\ |p_3 - p'_3| \leq \varepsilon & \wedge \\
 & \multicolumn{2}{l|}{P_{\cdot p_1} \sim_\varepsilon P'_{\cdot p'_1} \ \wedge \ Q_{\cdot p_2} \sim_\varepsilon Q'_{\cdot p'_2}} \\
 & \multicolumn{2}{l|}{P_{\cdot p_1}, Q_{\cdot p_2}, P'_{\cdot p'_1}, Q'_{\cdot p'_2} \in \mathbf{pPat}} \\
\hline
\end{array}$$

Note that two different pieces of information are indistinguishable if they are associated with probabilistic parameters that are smaller than the given tolerance ε, i.e. in practice both of them are considered to be a black box.

For instance, according to such a notion of probabilistic similarity, the expressions $M = \{m\}_K$ and $N = \{n\}_{K'}$ are indistinguishable if – given $G = \emptyset$, $pP_M = m_{\cdot p_1}$, $pP_N = n_{\cdot p_2}$, and a fixed threshold ε – the probabilities p_1 and p_2 are equal or smaller than ε. Also the expressions $M = \{m\}_K$ and $N = \{m\}_{K'}$ are indistinguishable if $p_1 = p_{dec}(\{m\}_K)$ and $p_2 = p_{dec}(\{m\}_{K'})$ are similar (even if not exactly the same). In practice, if $|p_1 - p_2| \leq \varepsilon$, then $M \approx_\varepsilon^G N$.

Proposition 1. *Given $M, N \in$ **Exp** it holds that:*

$$M \approx^G N \qquad \Rightarrow \qquad M \approx_\varepsilon^G N \quad \forall \varepsilon \in [0, 1[.$$

Proof. *See [21].*

Proposition 2. *Given $M, N \in$ **Exp** it holds that:*

$$M \approx^G N \qquad \Leftrightarrow \qquad M \approx_0^G N.$$

Proof. *A trivial consequence of the definition of compatibility relation.*

Finally, note that the case $\varepsilon = 1$ is not considered, since it is trivial to see that in such a case it follows $\forall M, N \in$ **Exp** $M \approx_1^G N$.

4 Relating the Probabilistic and the Dolev-Yao Models

In this section we show how our notion of similarity is related to a classical Dolev-Yao equivalence relation defined in an environment where perfect cryptography is assumed. In particular, given a notion of ideal encryption, we will show that two expressions are equivalent within a Dolev-Yao model that relies on perfect cryptography if and only if the two expressions are probabilistically similar under the ideal encryption assumption.

4.1 Equivalence Within Perfect Cryptography

We start by introducing a notion of equivalence for cryptographic expressions that relies on the perfect cryptography assumption. Such a notion is inspired by that defined in [2]. First, we define a variant of the set of expressions, called **Pat**, which does not contain ciphertexts and includes the new symbol $\otimes$ (representing a ciphertext that cannot be decrypted by the adversary).

$P, Q ::=$	patterns
K	key (for $K \in \mathbf{Keys}$)
m	string (for $m \in \mathbf{String}$)
(P, Q)	pair
$\otimes$	undecryptable text

Intuitively, a *pattern* is an expression that does not contain encrypted terms and that may contain some part that an adversary cannot decrypt. Now, we define a function p that, given a set of keys T and an expression M, computes the pattern that an adversary can obtain from M if the initial knowledge is the set of keys T.

$$\begin{array}{lll}
p(K, T) & = K & (\text{for } K \in \mathbf{Keys}) \\
p(m, T) & = m & (\text{for } m \in \mathbf{String}) \\
p((M, N), T) & = (p(M, T), p(N, T)) & \\
p(\{M\}_K, T) & = \begin{cases} p(M, T) & \text{if } K \in T \\ \otimes & \text{otherwise} \end{cases} &
\end{array}$$

Then, given an initial knowledge G, we define function $pat^G(M)$, which expresses the pattern obtained from an expression M by exploiting the knowledge G, as $pat^G(M) = p(M, initKeys((M, G)))$. For example, if G is empty, $pat^G((\{\{K_1\}_{K_2}\}_{K_3}, K_3)) = (\otimes, K_3)$.

Finally, given an initial knowledge G, we say that two expressions are *equivalent* if they yield the same pattern.

Definition 3. $M \cong^G N \quad \Leftrightarrow \quad pat^G(M) = pat^G(N)$.

For example, if G is empty, we have $(\{\{K_1\}_{K_2}\}_{K_3}, K_3) \cong (\{\{m\}_{K_1}\}_{K_3}, K_3)$ since both expressions yield the pattern $(\otimes, K_3)$.

4.2 Ideal Encryption

The notion of *ideal encryption* intuitively assumes that it should be hard for the adversary to decrypt a message ciphered with an unknown key. In other words, the probability of breaking an encrypted message that cannot be derived in the classical Dolev-Yao model should be *small*. We formalize the concept of small probabilities by introducing the definition of *negligible* function (see, e.g., [11]).

Definition 4. *A function* $f : \mathbb{N} \rightarrow \mathbb{R}$ *is negligible, if for any polynomial* q, $\exists \eta_0 \in \mathbb{N} : f(\eta) \leq \frac{1}{q(\eta)} \quad \forall \eta > \eta_0$.

Then, the *ideal encryption* hypothesis assumes that p_{dec} must be a negligible function.

Definition 5. *An encryption scheme is ideal if and only if*

$\forall \{N\}_K \in \mathbf{Exp}, \forall G \in \mathbf{Exp} : G \not\mapsto K, \forall$ *polynomial* $q : \exists \eta_0 \in \mathbb{N}$ *such that* $p_{dec}(\{N\}_K, G) \leq \frac{1}{q(\eta)} \quad \forall \eta > \eta_0$.

As a consequence, if the assumption of ideal encryption holds, from the definition above we also have that $\forall \mathcal{A}$ and $\forall G \in \mathbf{Exp} : G \not\mapsto K$:

$$Pr[K \leftarrow \mathcal{A}(\{N\}_K, G)] \leq \frac{1}{q(\eta)} \quad \forall \eta > \eta_0.$$

By following an approach also used in [22], we show that a result holding in the perfect cryptography scenario also holds in our probabilistic model (and vice versa) if the ideal encryption assumption is taken.

Theorem 1. *Given* $M, N \in \mathbf{Exp}$*, if the assumption of ideal encryption holds for a polynomial* q *and a natural* η_0*, then, for each* $\eta > \eta_0$*:*

$$M \cong^G N \quad \Leftrightarrow \quad M \approx^G_\varepsilon N \quad \forall \varepsilon \in]\frac{1}{q(\eta)}, 1 - \frac{1}{q(\eta)}[.$$

Proof. $\Rightarrow$) *A proof of the statement is in [21].*
$\Leftarrow$) *The statement derives by structural induction on the expression* M *and by observing that, by hypothesis,* $M \approx^G_\varepsilon N \Rightarrow pP^G_M \sim_\varepsilon pP^G_N$*. In the following, we denote by* T_M *the set* $initKeys((M, G))$ *and by* T_N *the set* $initKeys((N, G))$*.*

1. $pP^G_M \sim_\varepsilon pP^G_N = K_{\cdot 1} \qquad K \in \mathbf{Keys}$
   $\Rightarrow$
   $p(M, T_M) = p(N, T_N) = K \Rightarrow pat^G(M) = pat^G(N) \Rightarrow M \cong^G N$
2. $pP^G_M \sim_\varepsilon pP^G_N = m_{\cdot 1} \qquad m \in \mathbf{String}$
   $\Rightarrow$
   $p(M, T_M) = p(N, T_N) = m \Rightarrow pat^G(M) = pat^G(N) \Rightarrow M \cong^G N$
3. $pP^G_M = P_{\cdot p} \sim_\varepsilon Q_{\cdot p'} = pP^G_N \qquad p, p' \leq \varepsilon \quad P_{\cdot p}, Q_{\cdot p'} \in \mathbf{pPat}$
   $\Rightarrow$
   $p(M, T_M) = p(N, T_N) = \otimes \Rightarrow pat^G(M) = pat^G(N) \Rightarrow M \cong^G N$
4. $pP^G_M = (pP^G_M(L_1, \emptyset), pP^G_M(L_2, \emptyset))_{\cdot 1} \sim_\varepsilon (pP^G_N(L'_1, \emptyset), pP^G_N(L'_2, \emptyset))_{\cdot 1} = pP^G_N \Rightarrow$
   $pP^G_M(L_1, \emptyset) \sim_\varepsilon pP^G_N(L'_1, \emptyset) \ \wedge \ pP^G_M(L_2, \emptyset) \sim_\varepsilon pP^G_N(L'_2, \emptyset)$
   $\Rightarrow$ *by induction hypothesis*
   $p(L_1, T_M) = p(L'_1, T_N) \ \wedge \ p(L_2, T_M) = p(L'_2, T_N) \Rightarrow$
   $p(M, T_M) = (p(L_1, T_M), p(L_2, T_M)) = (p(L'_1, T_N), p(L'_2, T_N)) = p(N, T_N) \Rightarrow$
   $pat^G(M) = pat^G(N) \Rightarrow M \cong^G N$

Under the assumption of ideal encryption, the four cases above include all the interesting situations in which two probabilistic patterns are compatible according to $\sim_\varepsilon$*. In particular, the condition* $|p - p'| \leq \varepsilon$ *is always true if* $p, p' \neq 1$*. Indeed, thanks to the ideal encryption assumption stating that* $p, p' \leq \frac{1}{q(\eta)}$*, if* $p, p' \neq 1$*, we have that* $p, p' < \varepsilon$*. Therefore, the three cases* $K_{\cdot p} \sim_\varepsilon K_{\cdot p'}$*,* $m_{\cdot p} \sim_\varepsilon m_{\cdot p'}$ *and* $(P_{\cdot p_1}, Q_{\cdot p_2})_{\cdot p} \sim_\varepsilon (P'_{\cdot p'_1}, Q'_{\cdot p'_2})_{\cdot p'}$ *collapse into the case* $P_{\cdot p} \sim_\varepsilon Q_{\cdot p'}$*,* $p, p' \leq \varepsilon$ *(case 3 of the proof of* $\Leftarrow$*)), when* $p, p' \neq 1$*.*

Finally, note that we did not consider the cases in which $P{\cdot}_p \sim_\varepsilon Q{\cdot}_{p'}$ *for some* $P, Q \in$ **pPat** *with* $p = 1$ *(*$p \neq 1$*) and* $p' \neq 1$ *(*$p' = 1$*). In such cases, the condition* $|1-p'| \leq \varepsilon$ *(*$|1-p| \leq \varepsilon$*) does not hold, since, by the ideal encryption assumption,* $p' \leq \frac{1}{q(\eta)}$ *(*$p \leq \frac{1}{q(\eta)}$*) and, by the premises of the theorem,* $\varepsilon < 1 - \frac{1}{q(\eta)}$*. As a consequence, it is impossible to find a case in which* $P{\cdot}_p \sim_\varepsilon Q{\cdot}_{p'}$ *for some* $P, Q \in$ **pPat** *with* $p = 1$ *(*$p \neq 1$*) and* $p' \neq 1$ *(*$p' = 1$*).* ∎

5 Secrecy Against the Computational Adversary

In this section we introduce a notion of secrecy of some information within a given expression. Consider for example the expression $M = (\{m\}_K, \{n\}'_K)$. We are interested in evaluating whether the expression M *maintains* a given secret m assuming that the expression G models the actual knowledge of an adversary[3]. In particular, we are also interested in evaluating the degree of secrecy of m within M. Intuitively, we observe that a certain secret m is private in M if the expression N, obtained by substituting every occurrence of m in M with $m' \neq m$, is similar to M. More formally, we provide the following definition.

Definition 6. *Given a knowledge modeled by the expression* G*, a certain secret* α *such that* $\alpha \in$ **Keys** *or* $\alpha \in$ **String***, a parameter* $\varepsilon \in [0, 1[$ *and an expression* $M \in$ **Exp** *such that* α *occurs in* M*, we say that* α *is* ε_G-secret *in* M *iff the expression* N *obtained by substituting every occurrence of* α *in* M *with the key* $K \neq \alpha$ *(if* $\alpha \in$ **Keys***) or with the string* $m \neq \alpha$ *(if* $\alpha \in$ **String***) is such that* $M \approx_\varepsilon^G N$*.*

Intuitively, Definition 6, inspired by [1], states that a certain secret α is private within an expression M if an adversary is not able to distinguish M from the expression N obtained by substituting in M every occurrence of α with $\alpha' \neq \alpha$. In a sense, that means the adversary is not able to extract α from M with a probability greater than ε. Therefore, if α is ε_G*-secret* in M, we can deduce that the adversary with knowledge G can extract α from M with a success probability equal or smaller than ε.

Example 2. Consider the expression $M = (\{m\}_{K_1}, \{K\}_{K_2})$ and a knowledge $G = K_1$.

On the one hand, we obviously have that m is not ε_G-secret in M for any $\varepsilon \in [0, 1[$. Given $N = (\{m'\}_{K_1}, \{K\}_{K_2})$ we have that $M \not\approx_\varepsilon^G N$, in fact $pP_M^G = (m{\cdot}_1, K{\cdot}_p)$ where $p = p_{dec}(\{K\}_{K_2}, (M, G)) = p_{dec}(\{K\}_{K_2}, (N, G))$ and $pP_N^G = (m'{\cdot}_1, K{\cdot}_p)$. Since $m \neq m'$ we obviously have that $pP_M^G \not\sim_\varepsilon pP_N^G$ so that, in practice, $M \not\approx_\varepsilon^G N$.

On the other hand, we have that K is ε_G-secret in M for any $\varepsilon \in [p, 1[$. Given $N = (\{m\}_{K_1}, \{K'\}_{K_2})$ we have that $M \approx_\varepsilon^G N$ for any $\varepsilon \in [p, 1[$, in

[3] The expression G models, for example, the sequence of messages exchanged within the network until a certain moment, and the set of keys known by the adversary.

fact $pP_M^G = (m._1, K._p)$ and $pP_N^G = (m._1, K'._p)$. Since $\varepsilon \geq p$ we have that the adversary is not able to distinguish K from K' ($pP_M^G \sim_\varepsilon pP_N^G$), so that in practice we have that $M \approx_\varepsilon^G N$.

5.1 An Application

We apply the notion of secrecy within an expression in a very simple real case. Consider a protocol where a server S could be asked to generate a secret key and then send it back to the entity A that applied the request. The server also monitors and keeps track of all the messages exchanged in the network.

Assuming that an authentication phase precedes the protocol, we denote with K_{SA} a key shared between the server S and the entity A. Finally, we use t to denote a time stamp. The protocol can be described as follows (with the standard notation $A \rightarrow B : Msg$ we denote a message Msg sent by A and received by B):

$$\begin{array}{l} 1.\ A \rightarrow S : \{request, A, S, t\}_{K_{SA}} \\ 2.\ S \rightarrow A : \{K, S, A, t\}_{K_{SA}} \end{array}$$

where K is the secret key generated by the server.

We now translate the messages exchanged by the protocol into cryptographic expressions, by assuming that $K_{SA}, K \in$ **Keys** and that $request, A, S, t$ correspond to $r, a, s, t \in$ **String**, respectively. Hence, we have that the protocol exchanges the following expressions:

$$\begin{array}{l} 1.\ \{(r, ((a, s), t))\}_{K_{SA}} \\ 2.\ \{(K, ((s, a), t))\}_{K_{SA}} \end{array}$$

Now, assume that all the messages exchanged in the network are modeled by the formal expression G. Then, we apply our notion of secrecy within expressions in order to check whether the expression $\{(K, ((s, a), t))\}_{K_{SA}}$ ensures a given degree ε of secrecy for K. To this end, what the server needs to do is to check whether K is ε_G-secret in $\{(K, ((s, a), t))\}_{K_{SA}}$. The parameter ε is fixed by the server and represents the security threshold needed to guarantee the secure execution of the protocol. Note that, as the traffic of information within the network increases and the amount of messages ciphered with the shared key K_{SA} gets larger, the server may not guarantee the ε_G-secrecy anymore. Our notion of secrecy within expressions is able to capture situations of this kind. Therefore, if at a certain instant of time K is not ε_G-secret in $\{(K, ((s, e), t))\}_{K_{SA}}$ anymore, the server may, for example, activate a procedure generating a new shared key with the entity A and then send the secret to A encrypted with the fresh key.

6 Related Work

The treatment of cryptographic operations within formal models is covered by a quite large body of literature, but most of these efforts do not consider cryptographic operations in an imperfect cryptography scenario.

This work represents a step toward the definition of a formal language with cryptographic primitives and conditional statements for analyzing both unwanted disclosure of data due to the nature of the protocols and information leakage due to the nature of the cryptographic means. In the literature, both probability and computational complexity are studied in formal settings.

Process algebra and computational view of cryptography are combined in [14] where, in the setting of a subset of asynchronous π-calculus, an asymptotic notion of probabilistic equivalence is defined. The observational equivalence defined in terms of such a notion can be related to polynomial time statistical tests, i.e. equivalent processes are indistinguishable from the viewpoint of polynomial time adversaries. Security is then stated in terms of indistinguishability between the protocol under analysis and an idealized protocol specification. More recently, a definition of probabilistic noninterference which includes a computational case has been defined in [5] in the setting of asynchronous probabilistic reactive systems. In particular, computational noninterference means that the advantage of the external observer (which interacts with the system under analysis) for a correct guess of the interfering adversary behavior is a negligible function. A formal notion of computational indistinguishability is also defined in [13] on the basis of a simple model where public outputs are observed in order to infer the content of secret inputs. Finally, [22] compares the classical Dolev-Yao adversary with an enhanced computational adversary which can guess the key for decrypting an intercepted message (albeit only with negligible probability). The two adversaries are shown to be equivalent with respect to a secrecy property. Moreover, in [22] the authors define a function similar to our p_{dec} in order to model the probability for a computational adversary of guessing a key. However they abstract away from the particular ciphertext in which the key to be guessed is used as the ciphering key, and from the knowledge the adversary gets. As we do, they also abstract away from how the probability $p_{dec}(\{m\}_K)$ could be computed.

We finally point out that probabilistic notions of security as well as approximate security properties can be found in the literature (see, e.g., [10, 7, 4, 3]), but they do not relate probability and cryptographic primitives.

7 Conclusion

In this paper we proved that a standard notion of Dolev-Yao adversary equates the expressive power of a computational adversary in the case ideal encryption is assumed. This is done in a formal framework where indistinguishability among cryptographic expressions is defined by means of a notion of probabilistic similarity taking into account computational poly-time adversaries.

The formal comparison among cryptographic expressions and its application to the verification of security properties represents an important step towards the definition of a formal framework for modelling cryptographic protocols and analyzing their robustness against malicious parties. In particular, the robustness of a system can be evaluated both in terms of absence of (probabilistic) covert channels and in terms of effectiveness of cryptanalysis attacks. While the first kind of attack has been deeply analyzed in the literature (see, e.g., [7, 4] and the references therein), in this paper we concentrated on the second type of security problem and we proposed an approach, which, as a further line of investigation, aims at putting the basis for a formal framework where both families of security flaws can be attacked in an integrated way.

References

1. M. Abadi, A. D. Gordon, *"A Calculus for Cryptographic Protocols: The Spi Calculus"*, Information and Computation, 148(1):1–70, 1999.
2. M. Abadi, P. Rogaway, *"Reconciling Two Views of Cryptography (The Computational Soundness of Formal Encryption)"*, in Proc. of 1st IFIP Int. Conf. on Theoretical Computer Science, Springer LNCS 1872:3-22, 2000.
3. A. Aldini, M. Bravetti, A. Di Pierro, R. Gorrieri, C. Hankin, H. Wiklicky, *"Two Formal Approaches for Approximating Noninterference Properties"*, Foundations of Security Analysis and Design II, R. Focardi and R. Gorrieri, eds., Springer LNCS 2946:1-43, 2004.
4. A. Aldini, M. Bravetti, R. Gorrieri, *"A Process-algebraic Approach for the Analysis of Probabilistic Non-interference"*, Journal of Computer Security, vol. 12(2):191-245, IOS Press, 2004.
5. M. Backes, B. Pfitzmann, *"Computational Probabilistic Non-interference"*, in Proc. of 7th European Symposium on Research in Computer Security, Springer LNCS 2502:1-23, 2002.
6. A. Datta, R. Kusters, J. C. Mitchell, A. Ramanathan, V. Shmatikov, *"Unifying Equivalence-Based Definitions of Protocol Security"*, in Proc. of Workshop on Issues in the Theory of Security, WITS'04, 2004.
7. A. Di Pierro, C. Hankin, H. Wiklicky, *"Approximate Non-Interference"*, in Proc. of 15th Computer Security Foundations Workshop, IEEE CS Press, pp. 1-17, 2002.
8. D. Dolev, A. Yao, *"On the Security of Public-key Protocols"*, IEEE Transactions on Information Theory, 29:198-208, 1983.
9. A. Durante, R. Focardi, R. Gorrieri, *"A Compiler for Analysing Cryptographic Protocols Using Non-Interference"*, ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 9(4):489-530, 2000.
10. J. W. Gray III, *"Toward a Mathematical Foundation for Information Flow Security"*, Journal of Computer Security, 1:255-294, 1992.
11. J. Herzog, *"A Computational Interpretation of Dolev-Yao Adversaries"*, in Proc. of 3rd Int. Workshop on Issues in the Theory of Security (WITS'03), 2003.
12. R. A. Kemmerer, *"Analyzing Encryption Protocols using Formal Verification Techniques"*, IEEE Journal on Selected Areas in Communications, 7(4):448-457, 1989.
13. P. Laud, *"Semantics and Program Analysis of Computationally Secure Information Flow"*, in Proc. of 10th European Symposium on Programming (ESOP'01), Springer LNCS 2028:77-91, 2001.
14. P. Lincoln, J. C. Mitchell, M. Mitchell, A. Scedrov, *"A Probabilistic Poly-Time Framework for Protocol Analysis"*, in Proc. of 5th ACM Conf. on Computer and Communications Security, ACM Press, pp. 112-121, 1998.
15. D. Micciancio, B. Warinschi, *"Completeness Theorems for the Abadi-Rogaway Language of Encrypted Expressions"*, in 2nd ACM SIGPLAN and IFIP WG 1.7 Workshop on Issues in the Theory of Security (WITS'02), Portland (OR), 2002.

16. J. K. Millen, S. C. Clark, S. B. Freedman, *"The Interrogator: Protocol Security Analysis"*, IEEE Transactions on Software Engineering, SE-13(2):274-288, 1987.
17. L. C. Paulson, *"The Inductive Approach to Verifying Cryptographic Protocols"*, Journal of Computer Security, 6(1-2):85-128, 1998.
18. A. Ramanathan, J. Mitchell, A. Scedrov, V. Teague, *"Probabilistic Bisimulation and Equivalence for Security Analysis of Network Protocols"*, in Proc. of 7th Int. Conf. on Foundations of Software Science and Computation Structures (FOSSACS'04), Springer LNCS 2987:468-483, 2004.
19. S. Schneider, *"Security Properties and CSP"*, in IEEE Symposium on Security and Privacy, IEEE CS Press, pp. 174-187, 1996.
20. A. Troina, A. Aldini, R. Gorrieri, *"A Probabilistic Formulation of Imperfect Cryptography"*, in Proc. of 1st Int. Workshop on Issues in Security and Petri Nets, WISP'03, 2003.
21. A. Troina, A. Aldini, R. Gorrieri, *"Approximating Imperfect Cryptography in a Formal Model"*, in Proc. of Mefisto Project Final Workshop, Elsevier ENTCS, to appear, available at `http://mefisto.web.cs.unibo.it/pubbl.html`.
22. R. Zunino, P. Degano, *"A Note on the Perfect Encryption Assumption in a Process Calculus"*, in Proc. of 7th Int. Conf. on Foundations of Software Science and Computation Structures (FOSSACS'04), Springer LNCS 2987:514-528, 2004.

For-LySa: UML for Authentication Analysis[⋆]

Mikael Buchholtz[1], Carlo Montangero[2],
Lara Perrone[2], and Simone Semprini[3,⋆⋆]

[1] Informatics and Mathematical Modelling,
Technical University of Denmark, Richard Petersens Plads,
DTU-bldg. 321, DK-2800 Kgs. Lyngby, Denmark
mib@imm.dtu.dk

[2] Dipartimento di Informatica, Università di Pisa,
Via F. Buonarroti 2 I-56127 Pisa, Italy
monta@di.unipi.it

[3] Automated Reasoning Systems Division, ITC-IRST,
Via Sommarive 18, I-38050 Povo – Trento, Italy
semprini@itc.it

Abstract. The DEGAS project aims at enriching standard UML-centred development environments in such a way that the developers of global applications can exploit automated formal analyses with minimal overhead. In this paper, we present For-LySa, an instantiation of the DEGAS approach for authentication analysis, which exploits an existing analysis tool developed for the process calculus LySa. We discuss what information is needed for the analysis, and how to build the UML model of an authentication protocol in such a way that the needed information can be extracted from the model. We then present our prototype implementation and report on some promising results of its use.

1 Introduction

Many years of research in formal methods have resulted in a wealth of analysis tools that, in theory, may assist software designers in the development of high quality products. In practice, however, these tools are often hard to use for non-experts and their direct, practical impact is therefore limited.

The overall aim of this paper is to illustrate that formal analysis tools can be used directly by designers of applications for global computing. To this end, we follow the approach of the DEGAS project where the idea, as illustrated in Figure 1, is to let developers use their own *development environment* while the formal analysis takes place in its own *verification environment*. More precisely, the development environment will be the Unified Modelling Language (UML)

[⋆] This work is partially funded by the Information Society Technologies programme of the European Commission, Future and Emerging Technologies, under the IST-2001-32072 project DEGAS.

[⋆⋆] This work was carried out when Simone Semprini was at the Dipartimento di Informatica, Università di Pisa

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 93–106, 2004.

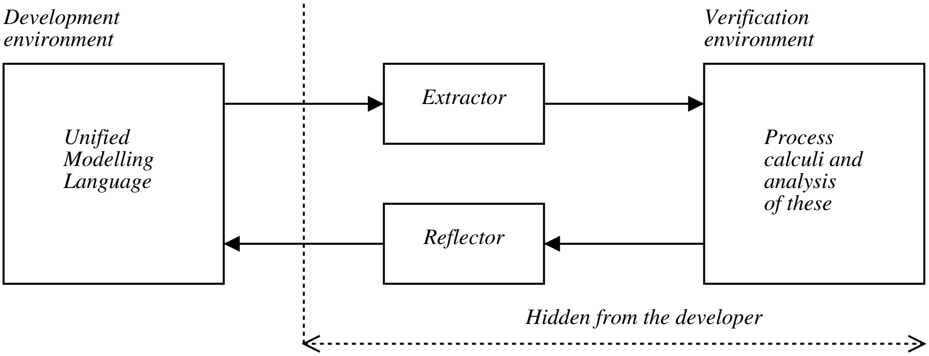

Development environment
Unified Modelling Language
Extractor
Reflector
Verification environment
Process calculi and analysis of these
Hidden from the developer

Fig. 1. Overview of the DEGAS approach to analysis of system design

that with is recent popularity in industry has a direct influence on many real world applications. The verification environment uses process calculi, which are *behavioural models* of systems, and the analysis of these calculi will therefore concentrate on behavioural aspects of systems. This nicely complements analyses of structural aspects such as well-typedness of object hierarchies and inter-diagram consistency, which are the kind of analysis that are typically carried out on UML today.

In order to perform analysis of the UML models in a verification environment the first step is to use an *extractor*, which extracts the parts of the model that will be relevant for the analysis and put these into the verification environment. After the analysis has been completed, the analysis result is made available to the developer using a *reflector*. To make this approach practical, from the point of view of the developer, the extractor, the analysis, and the reflector will all be automated and hidden from the developer. Thus, the developer will not need to know the finer details of these elements but may concentrate on the UML design of the system.

The main novelty of this paper, thus, is to illustrate that standard verification tools can indeed be used to analyse *security properties* of UML models. To this end, we give an overview of the For-LySa framework: an instantiation of the DEGAS approach targeted at designers of security critical applications that use network communication. Section 2 describes the application domain along with the security property of *authentication*, which will be checked in the verification environment based on the process calculus LySa [4]. Section 3 contains the UML modelling of applications using secure network communication including some additional features to cater for authentication analysis. Section 4 describes our prototype implementation of the For-LySa framework and, finally, Section 5 concludes the paper and comments on future and related work.

2 Security Protocols and Authentication

In a global computing environment, applications are typically distributed onto various host or principals, which communicate through a computer network. These communication patterns constitute a network protocol, which comprises the applications executed at the individual principals as well as their network communication. While we may rely on (some of) these principals to be trustworthy when executing the application, the network itself must be considered *unsafe* in the sense that hostile principals might tamper with the network messages.

The usual remedy to protect network protocols from intervention by malicious attackers is to apply cryptography so that parts of the messages may be kept outside the control of the attacker. In this paper we will illustrate how our approach works for a class of "classical" authentication protocols that use a shared server and symmetric key cryptography where the same key is used for encryption and decryption. This restricted, but representative, setup is chosen primarily to kept the extractor simple and we foresee no other significant challenges, neither for the UML modelling nor for the verification tool, in catering for more general scenarios. More precisely, we consider a network scenario in which a special principal, S, initially shares a unique key with each of the principals $A_1, \ldots, A_n$ and $B_1, \ldots, B_m$ and no other principals know these keys. The purpose of an authentication protocol that operate in this scenario is to allow two arbitrary principals A_i and B_j to be certain that a communication takes place between precisely these two principals and no one else.

To meet this goal, A_i, B_j, and S may, for example, engage in the following version of the Wide-mouthed-frog Protocol [5] (where we write $\{message\}_{key}$ for a message encrypted under a key):

$$\begin{array}{ll} 1.\ A_i \to S & : A_i, \{B_j, K\}_{K_{SA_i}} \\ 2.\ S \ \ \to B_j & : \{A_i, K\}_{K_{SB_j}} \\ 3.\ A_i \to B_j & : \{message\}_K \end{array}$$

In the protocol, the principal A_i generates a session key K, which it sends to the server, encrypted under the key K_{SA_i} shared only between S and A_i. The server decrypts the session key and forwards it encrypted to the B_j, which will afterwards be able to decrypt message 3 send by A_i. It is important to stress that *both* the message sequence *and* the internal action of each of the principals are equally important parts of the description of the protocol. Therefore, all these aspects will be modelled in UML in order to make the model amenable for a precise analysis of whether a protocol obtains its goal.

We focus on checking an authentication property, which loosely speaking says that "messages should end up in the right places". For example, if we consider the first message of the WMF, a property that we might like to have is that the message $A_i, \{B_j, K\}_{K_{SA_i}}$ should end up at S, only. However, nothing prevents an attacker from forwarding the two parts of the message to other principals than S so this property does not hold if the protocol is under influence of an attacker. Instead, the property we consider focuses on the parts of messages, which are not under the control of the attacker, namely, the parts where encryption has been applied. For example, the property that should hold for the first message of WMF is that the encrypted message $\{B_j, K\}_{K_{SA_i}}$ should be decrypted at S, only.

To specify the precise details in this kind of property we annotate the UML model giving a name, ℓ, to each point of encryption and each point of decryption. Furthermore, each encryption point will be annotated with which decryption points the encrypted message is intended to be decrypted at and, conversely, for decryption.

Our verification environment is based around the processes calculi LySa and a control flow analysis of this calculus [4]. LySa is a processes calculus in the π-calculus tradition [12] but tailored specifically to model central aspects of security protocols. The aim of the analysis is to tell whether authentication properties are satisfied *for all executions* of a LySa process executed in parallel with an arbitrary attacker process. The analysis will report *all possible breaches* of the authentication properties in an error component ψ: finding a pair (ℓ, ℓ') in ψ means that something encrypted at ℓ might be decrypted at ℓ' thereby breaking the specified authentication property.

The analysis works in form of a control flow analysis, which computes over-approximations to the behaviour of all executions of a LySa process. In particular, it computes over-approximations to the error component, which means that the analysis may report an error that is not actually there. However, it is proven in [4] that the analysis will *never report too few errors* and also illustrated that reporting too many errors is not at big problem in practice.

3 UML for Authentication Protocols

To model security protocols in a consistent way in UML, we define two UML profiles. They must be used when modelling specific protocols in order to make them amenable for analysis. First we present the profile *Static For-LySa* that describes how the concepts from the previous section, such as principals, keys, messages, etc., are modelled in UML. Next, we introduce a second profile, *For-LySa*, that is used to describe the dynamics of a protocol as well as the information needed for the analysis. Rather than presenting the profiles in tabular form, we present domain models, with the understanding that their classes and relations are the stereotypes in the profile. Note that we have two profiles to keep distinct what is actually needed to implement the protocol from what is additionally needed for the analysis.

3.1 The Static For-LySa Profile

The classes and associations in the class diagram on Figure 2 define the stereotypes in the profile Static For-LySa. The central classes in the diagram are Principal, Key, and Msg (for messages).

Keys can either be a SessionKey generated for each session or it can be the PrivateKey of a principal that is shared in advance with the Server. A Server is a special kind of principal that knows the private keys of all the other principals

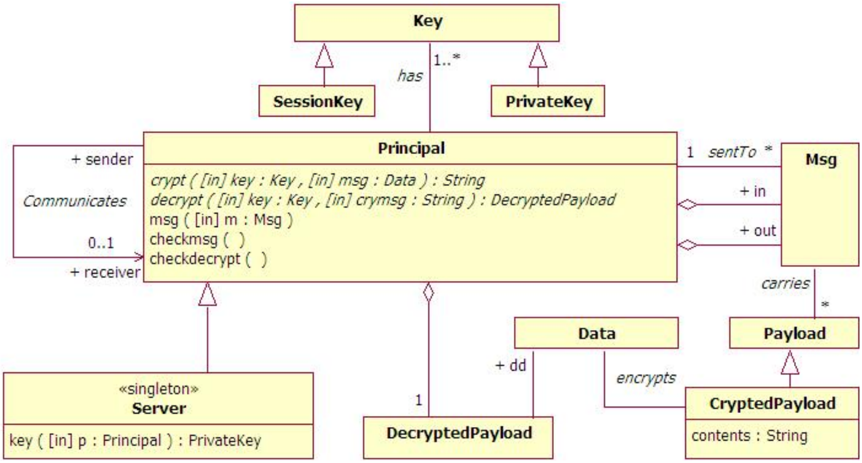


Fig. 2. The domain model

in the protocol. We represent this knowledge as an operation key() that, given a principal, returns its private key. The stereotype ⟪singleton⟫ ensures that in a given protocol specification only one server is used.

The principals communicate and exchange messages as shown by the communicates and sentTo associations. To express communication, the operation msg() can be invoked on the principal, which the message is sentTo. The specification of this operation is that it copies its argument into variable in of the receiver. For uniformity, and to ease the extraction process, we expect that the value of variable out is passed to msg(), i.e. that messages are put into this variable and then sent. Messages carry Payloads, some of which can be CryptedPayloads, which carry Data in their contents. In summary, whenever a principal contributes to a step of the protocol, it needs to keep track of two messages: an incoming message that is left by msg() in its in variable, and triggers the contribution, and the outgoing message that it builds in its out variable and then sends.

To specify a protocol, one needs to specify subtypes of Principal that introduce specific operations to set the outgoing messages, using the parts of the incoming ones as well as specific information held by the principal in private attributes. These operations should be introduced in a standardised way, that we will discuss in the sequel. The same applies to the operations that are needed to disassemble the incoming messages. However, there are some generic operations that can be used to build and open the messages. Some of these are left abstract, namely crypt() and decrypt(), since we leave the choice of the cryptographic algorithms open to further specialisation. In fact, the analysis treats encryption as abstract operations so we would not get more precise analysis results by specialising these operations further. The other two generic operations, checkMsg() and checkDecrypt() are null operations: they are introduced to allow the specifier to express the checks that need to be done on the incoming messages, and on the results of decryption actions, respectively. These checks can be conveniently

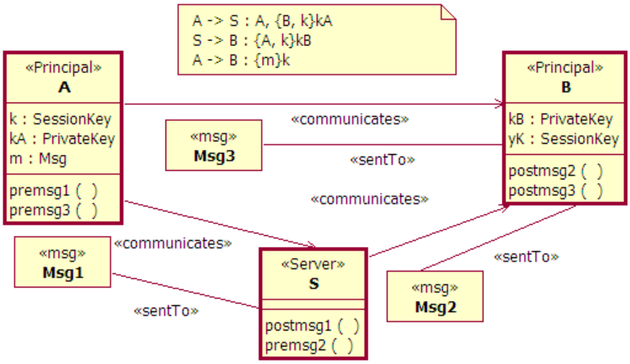


Fig. 3. The WMF protocol overview

expressed as constraints in the sequence diagram that describes the dynamics of the protocol in the For-LySa profile in Section 3.2. There, the protocol designer can use checkMsg() and checkDecrypt() as placeholders, to introduce the constraints on the incoming and decrypted data.

We assume that decrypt() leaves its result in variables of type DecryptedPayload that are named systematically for each relevant type of message, as we will see later. In this way, the checks and the operations that build new messages can exploit the results of decryptions in these variables. In a DecryptedPayload data can be accessed via dd (of type Data). The distinction of Data and DecryptedData is not strictly necessary, at this level of presentation, but it helps when extending the model to introduce the decorations needed for the analysis, as we will do in Section 3.2.

As an example of the use of Static For-LySa, we present the overview of the WMF protocol, described in the previous section. Figure 3 presents the structure of the protocol, showing the intended communications and the involved messages. The types of the principals are named A, B, and S for the initiator, the responder, and the server, respectively. The diagram also makes clear how the message types are named systematically, Msg1, Msg2, and Msg3, according to the order in which the messages are sent. The structure of each message is specified in distinct diagrams such as the one in Figure 4 that makes clear how the various parts of the message are named by systematically appending indexes to their types. Similar diagrams are introduced for the other message types.

In the diagram in Figure 3 we also introduce the names of the operations to build and dissect messages: for each message of type Msg$i$, there are operations premsg$i$ in the sender and postmsg$i$ in the receiver, to build and to open the message, respectively. The semantics of these operation will be specified by post-conditions. Finally, we introduce names for the local information of each principal, like private keys, session keys, and temporary storage, such as variable yK in principal B. This variable is needed by the responder to store the key, received in the second message, to be able to decrypt the third message sent by the initiator.

3.2 The For-LySa Profile

The UML view of the concepts that are needed to perform the authentication analysis are shown in Figure 5. In this figure, when we use the same names as in Figure 2 we denote entities that are specialising homonymous entities in the domain model. The other classes are new concepts, introduced for the analysis. These classes, and their associations and constraints define profile *For-LySa*.

First of all, each message carries with it the definition of the source principal, from which it is sent, along with the sink principal, which it should reach. Second, each encrypted payload is decorated with Cryptopoints. The idea is that, for each encrypted payload the annotations make explicit its origin, i.e. the point in the narration where the payload is encrypted, and its destinations, i.e. the set of the *intended* points of decryption. Similarly, for the decrypted data, the annotations make explicit the destination, i.e. the point in the narration where they are decrypted, and their *intended* origins, i.e. the set of expected places of encryption. Each crypto-point is a label, that will be associated to a single point of encryption (one of the premsg$i$) or decryption (postmsg$i$) in the dynamic view of the protocol.

As an example of the use of the For-LySa profile, Figure 6 presents the complete description of Msg1 including the decorations needed to specify the authentication property. The intended origin and destination of the encrypted part of the first message of WMF are specified to be Acp1 and Scp1, respectively. The stereotype ⟨⟨destIncludes⟩⟩ in Figure 6 is defined as the composition of the

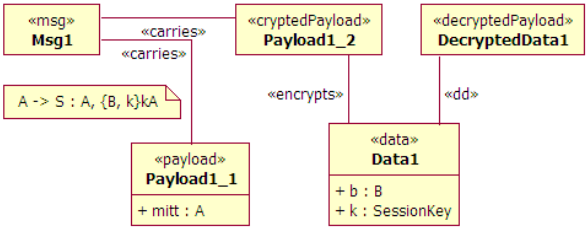


Fig. 4. The structure of Msg1

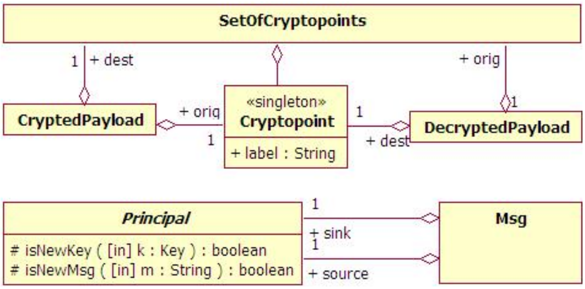


Fig. 5. The analysis model

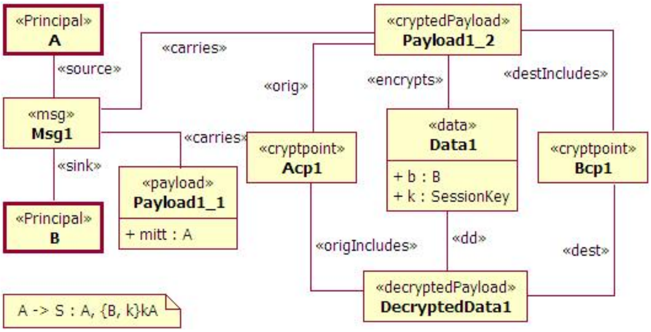


Fig. 6. The structure of Msg1 for the analysis

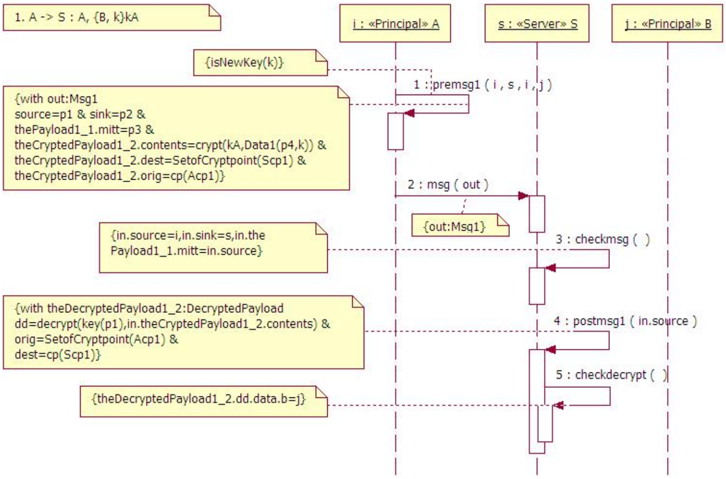

1. A -> S : A, {B, k}kA
i : «Principal» A
s : «Server» S
j : «Principal» B
{isNewKey(k)}
1 : premsg1 (i , s , i , j)
{with out:Msg1
source=p1 & sink=p2 &
thePayload1_1.mitt=p3 &
theCryptedPayload1_2.contents=crypt(kA,Data1(p4,k)) &
theCryptedPayload1_2.dest=SetofCryptpoint(Scp1) &
theCryptedPayload1_2.orig=cp(Acp1)}
2 : msg (out)
{out:Msg1}
{in.source=i,in.sink=s,in.the
Payload1_1.mitt=in.source}
3 : checkmsg ( )
{with theDecryptedPayload1_2:DecryptedPayload
dd=decrypt(key(p1),in.theCryptedPayload1_2.contents) &
orig=SetofCryptpoint(Acp1) &
dest=cp(Scp1)}
4 : postmsg1 (in.source)
5 : checkdecrypt ( )
{theDecryptedPayload1_2.dd.data.b=j}

Fig. 7. The sequence diagram for WMF

two aggregations from CryptedPayload to Cryptopoint in Figure 5. Similarly for ⟨⟨origIncludes⟩⟩. Similar diagrams describe the remaining two messages in the protocol.

To complete the WMF example, we need to address the dynamics of the protocol and this is done in a sequence diagram shown in Figure 7. The diagram adopts naming conventions consistent with those used in our scenario: the typical initiator object of type A is named i, the responder object of type B is named j, while the server object of type S is named s.

Each step in the protocol is divided into three sub-steps:

1. the sender packages the message,
2. the message is communicated,
3. the recipient processes the incoming message.

The third step is typically the most involved and includes tasks such as checking that the message format is correct, decrypting the parts intended for the current recipient, and storing the content of the

First, premsg1() builds the message in the out message of the sender. Second, msg() sends it to the recipient where it is stored as the in message. Finally, the recipient processes the received message by checking the message format with checkmsg(), decrypting the relevant part with postmsg1(), and ensuring the encrypted data also have the correct format with checkdecrypt(). When all these checks succeed the protocol continues similarly with the second and third messages (omitted for sake of space).

Operation msg(m: Msg) is polymorphic and accepts any message. However, the effective type of the message that is exchanged in each step, has to be specified as a constraint on the argument of msg(), as shown in the diagram. The other operations are not polymorphic, and have different names (and likely parameters) in each step. The signature of the operation is specified in the overview diagram of the analysis level, which is otherwise similar to Figure 3 and is not presented for space sake.

The operations are specified via post-conditions on the state of their principal. Post-conditions are attached to the operations as constraints, as shown in Figure 7 for premsgs and postmsgs. The natural place to attach these constraints would be the operations themselves, in the overview diagram, since they are the definition of the operation semantics. However, it is easier to follow the behaviour of the protocol having the post-condition attached to the call rather than to the definition, in another diagram.

We use a very simple language to write post-conditions. There is a record scope opener *á la* Pascal, for readability:

$$\mathsf{with}\, x : T\ \langle\text{condition}\rangle$$

means that x is constrained to be of type T, and that its selectors need not be prefixed by x in $\langle$condition$\rangle$. Conditions are conjunctions of equalities, where the left hand side identifies a field of the object, and the right side is an expression for its value. We use the standard dot notation to access object fields and to navigate along associations.

Expressions are built out of constructors, like SetofCryptpoint and Data$i$; operations like crypt and decrypt; variables, either parameters like p1 and p2, or locals of the principal that performs the operation, like out, kA, and k; and constants, i.e. the names of the objects, like i, and crypto-points labels, like Acp1.

The arguments to the constructors give values to the fields, in the order given in the diagrams that introduce them. Singleton crypto-point sets are built from the label, like in the last but one line of the post-condition for premsg1. The factory method cp builds Cryptopoint objects out of labels.

There are a couple of assumptions, with respect to keys:

- *private keys* can be freely used in the operations of the owner, since they are assumed to be initialised before the protocol starts;
- *session keys* must be initialised explicitly before they are used: for this purpose the For-LySa profile has the predicate isNewKey() (see Principal in

Figure 5) that can be used in a constraint before the first use of the variable. Using constraints leads to more concise diagrams than using explicit initialising operations, and has a straightforward mapping in a restriction operation in LySa.

As an example, the constraint attached to premsg1(i, s, i, j) in Figure 7 specifies that the value of the local variable out of initiator i of type A will be a message of the form $\mathsf{i}, \{\mathsf{j}, \mathsf{k}\}_{\mathsf{kA}}$, i.e. of the form of the message in the first step in WMF. Also, it describes the annotations of the authentication property, where Acp1 is the crypto-point associated with the encryption performed here, and Scp1 is associated to the corresponding decryption in postmsg1.

The post-condition attached to postmsg1 defines the effect of decrypting the message received by the server, in its variable theDecryptedPayload1_2. This is an example of the convention on decrypting actions: they leave their results in variables with names theDecryptedPayload*i*, where *i* is the same index of the corresponding CryptedPayload. In this example, the data are decrypted from the incoming message using a key passed as a parameter to the operation.

A number of checks on the messages have to be made explicit, to express dynamic constraints on the messages. These checks are expressed in UML as *invariant* constraints attached to the checking operations, in the sequence diagram. They are lists of equalities, with the syntax given above.

The source and sink of each message should be checked against the expected value. Additional checks depends on the specifics of the protocol, like the third condition attached to the third operation in Figure 7, which states that the clear payload must be equal to the message source, i.e. in this protocol each initiator can only speak for itself. Similarly, the next check, in the fifth operation, controls that the incoming responder (the b field in the encrypted payload) is indeed the intended one, namely j.

4 The For-LySa Prototype

We have developed a prototype implementation that can validate authentication properties of applications modelled in UML using the For-LySa profiles. The overall architecture of this *For-LySa prototype* follows the DEGAS approach on Figure 1.

In the For-LySa prototype, UML models are designed with Rational XDE version 1.5, and exported into XMI version 1.1, which is a standard, XML-based way to represent UML models.

The extractor is written in Java and takes as input the XMI representation of the UML model and delivers as output a corresponding LySa process annotated with the security properties specified in the UML model. The implementation of the extractor benefits from a generic Java library for writing extractors, which has been developed within the DEGAS project as part of the Java version of the PEPA Workbench [1]. The main operations of the extractor are: parsing the XMI file, building an intermediate representation, and finally generating a LySa process.

The verification tool is implemented in Standard ML and is available for download on the Web [2]. It takes as input a parameterised LySa process generated by the extractor and makes a finite instantiation of the scenario with $i = 1, \ldots, n$ and $j = 1, \ldots, m$ of principals A and B, respectively. The analysis, which is carried out on this finite instantiation of the scenario, returns an error component, ψ, containing pairs of crypto-points where the authentication property may be violated as explained in Section 2. The extractor has added indices ij to these crypto-points such that they will, in general, be of the form ℓ_{ij}.

Our current prototype does not include a reflector, as such. We simply, present the error component, ψ, to the developer. As illustrated in the next section, this information can directly be of use to the developer.

4.1 A Case Study

Using the For-LySa prototype on our running example, the WMF protocol, the analyser returns an empty error component, stating that no problems occur in any execution of the protocol — even in the presence of an attacker.

More precisely, the For-LySa prototype validates the protocol deployed in the scenario described in Figure 3 where additional attacker principals have access to the network. The For-LySa prototype validates that the authentication properties specified in the annotations to the UML model in Figure 6 and Figure 7 indeed hold for the WMF. That is, it validates that messages can only be successfully decrypted at the places specified in the annotations no matter what an attacker may try.

To illustrate the fine details that decides whether a protocol behaves correctly or not consider a slightly modified version of the WMF protocol where the first message is modified so that the identity of the responder is no longer encrypted. This affects, of course, the construction of the message in premsg1() as well as the checks made by the server in ckeckmsg(), postmsg1(), and ckeckdecrypt().

When we run the For-LySa prototype on this modified WMF protocol, it gives a non-empty error component, i.e. it reports that the authentication properties may be violated. Summarising the result in the error component, the analyser reports that something may go wrong because:

- something encrypted at $\mathsf{Acp1}_{ij}$ may be decrypted at $\mathsf{Scp1}$,
- something encrypted at $\mathsf{Acp2}_{ij}$ may be decrypted at $\mathsf{Bcp2}_{ij'}$ for $j \neq j'$, i.e. at a wrong responder,
- something encrypted at $\mathsf{Acp2}_{ij}$ may be decrypted at the attacker, and
- something encrypted at the attacker may be decrypted at $\mathsf{Bcp2}_{ij}$.

The first of these error messages signals that the encrypted part of the first message may be decrypted at Scp1 but that the server expected something that was not encrypted at $\mathsf{Acp1}_{ij}$. This may happen if the responders name, j, in first message is substituted by the name of another principal, say j', by the attacker and is possible in the modified WMF because the responders name is not encrypted. Next, in the second message, the server will forward the session key to the principal j' and consequently the thirds message may successfully be decrypted at j' i.e. at a wrong responder. This turns up as the second class of error message above.

There is a similar kind of attack, which allows the attacker to substitute his own name for the responders name in the first message and, consequently allows him to interact with the protocol as illustrated by the two last error messages.

Presenting these error messages to the developer allows him to pinpoint the precise places in the UML model where encryption fail to preserve authentication as indented. Of course, repairing the protocol on the basis of this information requires creativity on the part of the developer but the For-LySa prototype allows him to quickly validate whether modifications have the desired effect.

5 Conclusion

An overall aim of the work presented in this paper is to provide software developers with a high-level interface to formal analysis tools. The For-LySa framework specifically concentrates on using UML as the interface for developers of applications that contain secure network communication.

With the work presented here, we have reached a first milestone toward this aim — and with a positive result. We have provided the For-LySa UML profiles and illustrated how these may be used to model applications in our target domain. Furthermore, we are able to perform automated extraction and analysis, using the For-LySa prototype, thereby allowing developers to perform analysis of their UML models with no particular effort on their part.

Before the For-LySa framework can be tested extensively in the field, the loop of Figure 1 must be closed, to provide the relevant feed-back to the designer with a reflector. The problem is to find a convenient way to represent the illegal decryptions revealed by the analysis. This should be relatively straightforward except for the rather cumbersome task of automatically adding the error messages to the UML diagrams in a visually appealing manner.

5.1 Related Work

The overall aim of our work somewhat similar to the aim of frameworks such as Casper [11], CAPSL [6], CVS [7], and AVISS [3]. These frameworks all aim at providing developers of *security protocols* with high-level interfaces for formal analysis tools but unlike our approach they are based on ad-hoc notation, which describes protocols in an "$A \rightarrow B$: *message*"-style. On one hand, this may lead to more compact description of protocols than our but on the other hand we have all the advantages of using a general purpose modelling language.

On the technical side, the information found in protocol descriptions in the above frameworks is quite similar to the information captured in our message sequence diagrams. Also, in the extraction we find similarities, in particular with [11] that also has a target analysis formalism using a process calculus. The extraction made in [11] is, however, somewhat simpler than ours because its high-level language is designed so that is directly includes process calculi expression at convenient places.

An important effort that shares the DEGAS focus on the UML is centred on UMLsec [8]. This is a UML profile to express security-relevant information within the diagrams in a system specification, and on the related approach to secure system development [9]. UMLsec allows the designer to express recurring security requirements, like fair exchange, secrecy/confidentiality, secure information flow, secure communication link. Rules are given to validate a model against included security requirements, based on a formal semantics for the used fragment of UML, with a formal notion of adversary. This semantic base permits in principle to check whether the constraints associated with the UML stereotypes are fulfilled in a given specification. Work is ongoing to provide automatic analysis support, with an approach similar to that of DEGAS: [9] proposes to express protocols with sequence diagrams, translate them in first-order logic and then exploit standard theorem-provers, like e-SETHEO, to reveal potential attacks: [10]. The results of the analysis can be used to produce an attack scenario. In our opinion, For-LySa provides a more intuitive way to express authentication requirements that are less central in UMLsec: it should be worthwhile to assess the feasibility of the integration of the two approaches.

References

1. The Java edition of the Pepa workbench. Website hosted by School of Informatics, University of Edinburgh: http://homepages.inf.ed.ac.uk/s9905941/jPEPA/, May 2004.
2. LySa – a process calculus. http://www.imm.dtu.dk/cs_LySa, May 2004. Website hosted by Informatics and Mathematical Modelling, Technical University of Denmark.
3. A. Armando, D. Basin, M. Bouallagui, Y. Chevalier, L. Compagna, S. Mödersheim, M. Rusinowitch, M. Turuani, L. Viganò, and L. Vigneron. The AVISS security protocol analysis tool. In *CAV 2002*, volume 2404 of *Lecture Notes in Computer Science*, pages 349–353. Springer Verlag, 2002.
4. C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Automatic validation of protocol narration. In *Proceedings of the 16th Computer Security Foundations Workshop (CSFW 2003)*, pages 126–140. IEEE Computer Society Press, 2003.
5. M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, pages 18–36, 1990.
6. G. Denker, J. Millen, and H. Rueß. The CAPSL integrated protocol environment. Technical Report SRI-CLS-2000-02, SRI International, 2000.
7. A. Durante, R. Focardi, and R. Gorrieri. A compiler for analyzing cryptographic protocols using noninterference. *ACM Transactions on Software Engineering and Methodology*, 9(4):488–528, 2000.
8. J. Jürjens. UMLsec: Extending UML for secure systems development. In *UML 2002 – The Unified Modeling Language*, volume 2460 of *Lecture Notes in Computer Science*, pages 412–425, 2002.
9. J. Jürjens. *Secure Systems Development with UML.* Springer Verlag, 2004. To appear.

10. J. Jürjens and T. A. Kuhn. Automated theorem proving for cryptograpich protocols with automatic attack generation, 2004. Personal Communication.
11. G. Lowe. Casper: A compiler for the analysis of security protocols. *Journal of Computer Security*, 6(1):53–84, 1998.
12. R. Milner, J. Parrow, and D. Walker. A calculus of Mobile processes (I and II). *Information and Computation*, 100(1):1–77, 1992.

Performance Analysis of a UML Micro-business Case Study*

Katerina Pokozy-Korenblat, Corrado Priami, and Paola Quaglia

Dipartimento di Informatica e Telecomunicazioni, Università di Trento, Italy

Abstract. This paper presents a technique to carry out performance analysis of UML specifications. We consider UML specifications composed of activity, sequence and deployment diagrams. Specifications are translated into the stochastic π-calculus, and quantitative analysis is then performed via the BioSpi tool. The approach is applied to a web-based Micro-business case study.

1 Introduction

The Unified Modelling Language (UML) [2] is an a-posteriori industrial standard for high-level design of software systems. The quantitative evaluation of different design alternatives, and hence the understanding of system performance can aid the design of complex systems. In this respect, it would be useful to propose to UML modelers an automated tool that allows performance analysis of UML specifications and nonetheless is free from the typical complexities of the analyses carried out over formal descriptions.

Our work goes in the direction sketched above. The approach consists in first compiling UML specifications into π-calculus intermediate representations,

and then carrying out performance analysis by the BioSpi tool [1]. We show the feasibility of our approach on a web-based micro-business case study [3] developed by Motorola within the DEGAS project [4].

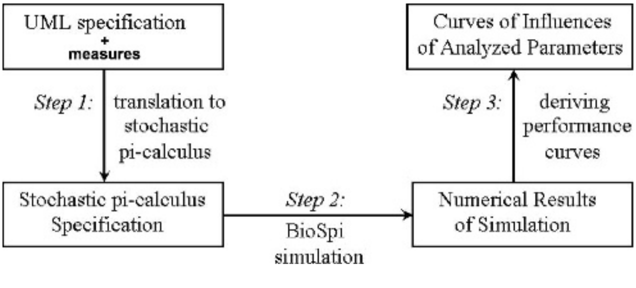

UML specification
+
measures
Step 1:
translation to stochastic pi-calculus
Stochastic pi-calculus Specification
Step 2:
BioSpi simulation
Numerical Results of Simulation
Step 3:
deriving performance curves
Curves of Influences of Analyzed Parameters

Fig. 1. Performance analysis of UML specifications

* Work partially funded by the IST-2001-32072 project DEGAS.

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 107–126, 2004.

Figure 1 presents a schematic view of the approach. Starting from activity, sequence and deployment diagrams enriched with supplementary quantitative parameters we obtain a corresponding stochastic π-calculus specification (*Step 1*). This last specification is simulated using the BioSpi tool (*Step 2*). On the basis of the simulation results we derive performance curves characterizing the influence of the given quantitative measures (channel bandwidth, authentication time, and the like) over the behaviour of the global system (*Step 3*).

In Section 2 we introduce the UML specification of the web-based micro-business case study. For convenience the UML diagrams are collected all together in the appendix at the end of the paper. The definition of the stochastic π-calculus is given in Section 3. In Section 4 we present our translation from the UML specification to the stochastic π-calculus dealing with the translation of activity diagrams, the translation of sequence diagrams, the management of the scoping of names, and the proper setting of quantitative parameters. The paper ends with some of the results we got by carrying out performance analysis of the stochastic π-calculus specification of the UML case study by the BioSpi tool.

2 UML Specification of the Case Study

Here we describe the web-based micro-business case study presented by Motorola in the DEGAS project as an example of a typical global computing application [3]. This case study aims at designing an application to provide support to micro web-based business to those enterprises that do not possess the capability of developing proprietary solutions for e-business. The players within the networked system form a community of sellers and buyers. The system offers to the buyer easy access to a variety of information about available services, and simple peer-to-peer e-commerce mechanisms.

Within the whole case study, we identify as the most critical process w.r.t. performance evaluation, the activity of buyers. A common view of the e-commerce functionality on buyers point of view is given in the E-Commerce activity diagram (Fig. 8). It shows the main steps to be executed to perform a peer-to-peer e-commerce transaction. First a buyer selects a seller. The next step is a handshake between the seller and the buyer to open a new session and to exchange a private key that will be used for the whole session.

The handshake activity is detailed in the Handshake sequence diagram that is reported at the top of Fig. 12. The communication mainly involves two objects: the Security Manager of the buyer (SM1) and the Security Manager of the seller (SM2). Security Managers are devoted to the implementation of the security mechanisms necessary to guarantee the confidentiality of the exchanged information. The handshake is initiated by the Security Manager of the buyer. It retrieves the address of the seller from the Sellers Manager which is responsible for handling the information related to the set of sellers that have registered with the buyer. The Sellers Manager checks the validity of the pair login/password that has been provided.

Once the address of the seller is known, the buyer sends a message of handshake using the same encryption key as the one used in the last session. The seller retrieves the key to decrypt the message from the Buyers Manager (whose role is analogous to the one played by the Sellers Manager). Once the seller has got the decryption key, it can interpret the message as a handshake. So the seller generates a new key for the current session, and sends it to the buyer, by encrypting it with the last session key. As soon as the buyer receives the new key, the last session key is abandoned.

During the handshake the buyer has to authenticate with the seller. For authentication purposes (see Fig. 12, bottom) the pair login/password is encrypted with the session key and sent from the buyer to the seller. The Security Manager of the seller retrieves the pair login/password stored by the Buyers Manager, and checks it against the pair received from the buyer. The result of the authentication is then sent to the buyer.

After the authentication, the buyer can browse the selling list to get the available items, prepare his basket, and make an e-commerce transaction.

Some information relevant to performance analysis is presented in the deployment diagram (see Fig. 10). In particular, the nodes Seller_S and Buyer_B are connected through the physical channel WIRELESS associated with a specific transfer rate. Moreover, the activity PerformTransaction is associated with a rate for direct communications with a banking system. Also, rates are used to define the complexity of the cryptography algorithms exploited by the messages Encrypt and Decrypt.

The activity diagram E-Commerce (Fig. 8), the activity diagram Prepare-Basket (Fig. 9), the deployment diagram (Fig. 10), and the sequence diagrams Handshake (Fig. 12, top), Authenticate (Fig. 12, bottom), and SearchSellerList (Fig. 11) are the subset of the UML case study specification which we use for our performance analysis.

3 The Stochastic π-Calculus

A brief description of the stochastic π-calculus [8] follows. It is an extension of the π-calculus [6, 9] that allows the stochastic modeling of the evolution of communicating and mobile systems.

As in the π-calculus, we assume the existence of a countably infinite set of *names*, called $\mathcal{N}$ and ranged over by $a, b, \ldots$ with $\mathcal{N} \cap \{\tau\} = \emptyset$. Also, a set $\mathcal{A}$ of *agent identifiers* and ranged over by $A, A_1, \ldots$ is assumed to be given. *Processes* (denoted by $P, Q, R, \ldots \in \mathcal{P}$) are built from names according to the following syntax

$$P ::= \mathbf{0} \mid (\pi, r).P \mid P + P \mid P|P \mid (\nu x)P \mid [x = y]P \mid A(y_1, \ldots, y_n)$$

where π may be $x(y)$ for *input*, $\overline{x}\langle y \rangle$ for *output* (where x is the *subject* and y the *object*), or τ for *silent* moves. Hereafter, the trailing '$\mathbf{.0}$' is always omitted.

In the prefix (π, r), π is an atomic action in the π-calculus sense, and r is the single parameter of an exponential distribution that characterizes the stochastic behaviour of the activity corresponding to the prefix π.

The input prefix binds the name y in the prefixed process. Intuitively, some name y is received along the link named x. The output prefix does not bind the name y which is sent along x. The silent prefix τ denotes an action which is invisible to an external observer of the system. Summation denotes nondeterministic choice. The operator $\mid$ describes parallel composition of processes.

The restriction operator (νy) in $(\nu y)P$ is a binder for y with scope P: it declares that y is a private resource of P, as opposed to a global (or public) name. An occurrence of a name in a process is *free* if it is not within the scope of a binder for that name. The set of free names of a process P is denoted by $fn(P)$. The matching operator $[x = y]P$ is an `if-then` constructor: process P is activated only if $x = y$. Eventually, $A(y_1, \dots, y_n)$ represents an agent depending on the parameters $y_1, \dots, y_n$ (hereafter denoted $\tilde{y}$). Each agent identifier A has a unique defining equation of the form $A(y_1, \dots, y_n) = P$, where the y_i are distinct and $fn(P) \subseteq \tilde{y}$.

The formal semantics of the calculus is given by a set of *congruence laws*, that determine when two syntactic expressions are equivalent, and by an *operational semantics* consisting of *reduction rules* to define the dynamic evolution of the system.

The *structural congruence* $\equiv$ on processes is defined as the least congruence satisfying the following clauses:

- P and Q α-equivalent (they only differ in the choice of bound names) implies $P \equiv Q$.
- $(P \mid Q) \mid R \equiv P \mid (Q \mid R)$, $P \mid Q \equiv Q \mid P$, $P \mid \mathbf{0} \equiv P$.
- $(P + Q) + R \equiv P + (Q + R)$, $P + Q \equiv Q + P$, $P + \mathbf{0} \equiv P$.
- $(\nu x)(P \mid Q) \equiv P \mid (\nu x)Q$ if $x \notin fn(P)$, $(\nu x)(\nu y)P \equiv (\nu y)(\nu x)P$, $(\nu x)\mathbf{0} \equiv \mathbf{0}$.
- $[x = x]P \equiv P$.
- $A(\tilde{y}) \equiv P\{\tilde{y}/\tilde{x}\}$ if $A(\tilde{x}) = P$ is the unique defining equation of the constant A and where $\{\tilde{y}/\tilde{x}\}$ denotes the substitution of the free occurrences of x_i with y_i with change of bound names to avoid name clashes.

The dynamic behavior of a process is driven by a race condition. All activities enabled attempt to proceed, but only the fastest one succeeds. The fastest activity is different on successive attempts because durations are random variables. The continuity of the probabilistic distribution ensures that the probability that two activities end simultaneously is zero. Furthermore, exponential distributions enjoy the *memoryless* property: the time at which a certain transition occurs is independent of the time at which it ever occurred before. Therefore there is no need to record the time elapsed to reach the current state.

The reduction semantics of the stochastic π-calculus is as follows (actually we report here the biochemical variant because we are going to use its implementation BioSpi to carry out performance analysis).

$$(\ldots + (\overline{x}\langle z\rangle, r_b).Q) \mid ((x(y), r_b).P + \ldots) \xrightarrow{x, r_b \cdot 1 \cdot 1} Q \mid P\{z/y\}$$

$$\frac{P \xrightarrow{x, r_b \cdot r_0 \cdot r_1} P'}{P \mid Q \xrightarrow{x, r_b \cdot r_0' \cdot r_1'} P' \mid Q} \qquad \begin{cases} r_0' = r_0 + In_x(Q) \\ r_1' = r_1 + Out_x(Q) \end{cases}$$

$$\frac{P \xrightarrow{x, r_b \cdot r_0 \cdot r_1} P'}{(\nu\, x)P \xrightarrow{x, r_b \cdot r_0 \cdot r_1} (\nu\, x)P'}$$

$$\frac{Q \equiv P \quad P \xrightarrow{x, r_b \cdot r_0 \cdot r_1} P' \quad P' \equiv Q'}{Q \xrightarrow{x, r_b \cdot r_0 \cdot r_1} Q'}$$

A communication event is implemented by the three parameters r_b, r_0 and r_1, where r_b represents the rate of the firing action, and r_0 and r_1 are computed using the two functions In_x and Out_x defined below. These two functions inductively count the number of receive and send offers on channel x. They are defined as follows:

$$\begin{aligned} &In_x(\mathbf{0}) = 0 \\ &In_x(\sum_{i \in I} (\pi_i, r_i).P_i) = |\{(\pi_i, r_i) | i \in I \wedge \pi_i = x(y)\}| \\ &In_x(P_1|P_2) = In_x(P_1) + In_x(P_2) \\ &In_x((\nu\, z)P) = \begin{cases} In_x(P) & \text{if } z \neq x \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

Out_x is similarly defined, by replacing any occurrence of In with Out and the condition $\pi_i = x(y)$ with $\pi_i = \overline{x}\langle y\rangle$.

Notational Conventions. The following notational conventions are adopted below. We often omit agent parameters and write $A = P$ instead of $A(\tilde{y}) = P$, the intended meaning being that $\tilde{y} = fn(P)$.

The choice composition $P_1 + \ldots + P_n$ is written $\Sigma_{i=1\ldots n}\ P_i$.

When the parameter y of an input action $x(y)$ (output action $\overline{x}y$, resp.) is irrelevant we abbreviate the action to x ($\overline{x}$, resp.). Also, given a set of names $V = \{v_1, \ldots v_n\}$, we use either $(\nu V)P$ or $\nu(v_1, \ldots, v_n)P$ to mean $(\nu v_1) \ldots (\nu v_n)P$, and $A(V)$ to mean $A(v_1, \ldots v_n)$.

Moreover, we use the following shorthands. When dealing with closed terms, we indicate the rates of channels as decorations of the corresponding restriction operator, and write, e.g., $(\nu x[r])(\overline{x}y \mid x(y))$ for $(\nu x)((\overline{x}y, r) \mid (x(y), r))$. When rates are denoted in this way, the shorthand (νx) is used to mean that channel x is associated with an infinite rate.

Finally, we sometime present specifications using the syntax of the polyadic π-calculus, from which the corresponding monadic version can be obtained as explained in [6].

4 From UML to the Stochastic π-Calculus

In this section we describe how, given some assumptions on the rates of actions, UML specifications are translated into the formal notation of the stochastic π-calculus.

UML allows the description of a system, but lacks the machinery to formally express the correlation between different diagrams. To overcome this issue we make a few assumptions on the relation between diagrams. They are relative to naming and are meant to allow the identification of elements occurring in distinct diagrams and still representing the same event. In particular, we consider UML specifications consisting of a set of sequence, activity and deployment diagrams satisfying the following requirements:

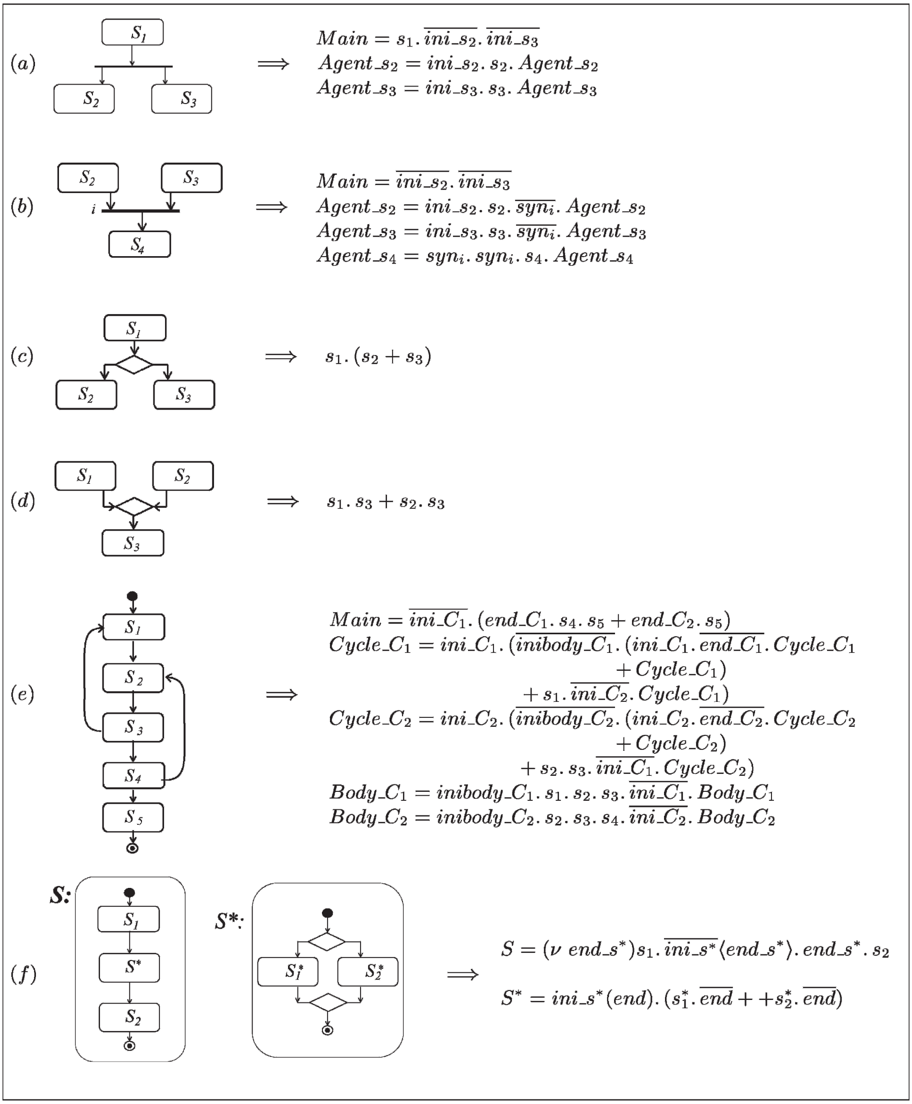
- If the name of an activity coincides with a name of some sequence or activity diagram, we consider it as a composite activity associated with that diagram.
- Each sequence diagram has to be associated with some activity.

In the following we first illustrate how activity and sequence diagrams are translated into plain π-calculus. Then we comment on the way measures are added to the processes resulting from this encoding, so leading to a stochastic π-calculus representation of the given UML specification. This is where deployment diagrams come into play. Indeed, measures related to performance can be given in various forms. They can be indicated as the rates of physical channels drawn in deployment diagrams, as the rates of the messages of sequence diagrams, as the rates of certain activities in the activity diagrams, or also as probabilities on choice constructions (multiple output, and branchings in activity and sequence diagrams).

4.1 Activity Diagrams

Our translation of activity diagrams into π-calculus processes is illustrated below. As a general rule, an activity is mapped into a π-calculus action, and a sequence of activities is translated into a sequential process composed by the corresponding actions. We explain in the following how forks, joins, and the other UML constructions are locally encoded into π-calculus processes. For synchronization purposes, the translations of some constructions make use of input and output actions over specialized channels. In the π-calculus process translating the global system, all of those channels are restricted and their scope is determined as explained below.

A **fork** construction (Fig. 2(a)) is represented by sending initialization signals in an arbitrary order to all the agents corresponding to the parallel control flows spawned by the fork. For each parallel control flow we define a recursive agent that is activated by receiving the initialization signal. For the sample in Fig. 2(a), e.g., we obtain the parallel composition of three processes: $Main$, $Agent\_s_2$, and $Agent\_s_3$. Agent $Main$, after executing the action s_1, unblocks the other two processes by sending signals over ini_s_2 and ini_s_3.



$$\begin{aligned}&Main = s_1.\,\overline{ini\_s_2}.\,\overline{ini\_s_3}\\ \Longrightarrow\quad &Agent\_s_2 = ini\_s_2.\,s_2.\,Agent\_s_2\\ &Agent\_s_3 = ini\_s_3.\,s_3.\,Agent\_s_3\end{aligned}$$

$$\begin{aligned}&Main = \overline{ini\_s_2}.\,\overline{ini\_s_3}\\ \Longrightarrow\quad &Agent\_s_2 = ini\_s_2.\,s_2.\,\overline{syn_i}.\,Agent\_s_2\\ &Agent\_s_3 = ini\_s_3.\,s_3.\,\overline{syn_i}.\,Agent\_s_3\\ &Agent\_s_4 = syn_i.\,syn_i.\,s_4.\,Agent\_s_4\end{aligned}$$

$$\Longrightarrow\quad s_1.\,(s_2 + s_3)$$

$$\Longrightarrow\quad s_1.\,s_3 + s_2.\,s_3$$

$$\begin{aligned}&Main = \overline{ini\_C_1}.\,(end\_C_1.\,s_4.\,s_5 + end\_C_2.\,s_5)\\ &Cycle\_C_1 = ini\_C_1.\,(\overline{inibody\_C_1}.\,(ini\_C_1.\,\overline{end\_C_1}.\,Cycle\_C_1\\ &\qquad\qquad\qquad\qquad + Cycle\_C_1)\\ \Longrightarrow\quad &\qquad\qquad + s_1.\,\overline{ini\_C_2}.\,Cycle\_C_1)\\ &Cycle\_C_2 = ini\_C_2.\,(\overline{inibody\_C_2}.\,(ini\_C_2.\,\overline{end\_C_2}.\,Cycle\_C_2\\ &\qquad\qquad\qquad\qquad + Cycle\_C_2)\\ &\qquad\qquad + s_2.\,s_3.\,\overline{ini\_C_1}.\,Cycle\_C_2)\\ &Body\_C_1 = inibody\_C_1.\,s_1.\,s_2.\,s_3.\,\overline{ini\_C_1}.\,Body\_C_1\\ &Body\_C_2 = inibody\_C_2.\,s_2.\,s_3.\,s_4.\,\overline{ini\_C_2}.\,Body\_C_2\end{aligned}$$

$$\begin{aligned}&S = (\nu\ end\_s^*)s_1.\,\overline{ini\_s^*}\langle end\_s^*\rangle.\,end\_s^*.\,s_2\\ \Longrightarrow\quad &\\ &S^* = ini\_s^*(end).\,(s_1^*.\,\overline{end} + +s_2^*.\,\overline{end})\end{aligned}$$

Fig. 2. Translation of elements of activity diagrams (fork, join, branch, merge, cycle, composite activity)

To translate a **join** construction (Fig. 2(b)) the ingoing parallel control flows are synchronized with the agent representing the outgoing edge to unblock the main activity that follows the join node (s_4 in the figure). Called i the join node, the special channel syn_i is introduced to force such synchronization.

A **branch** construction (Fig. 2(c)) is translated as the sum of the alternative activities, prefixed by the activity leading to the branch (s_1 in the example).

Multiple output transitions, whose semantics is similar to that of the branch construction, are encoded in an analogous way.

To translate a **merge** construction (Fig. 2(d)), we compose via nondeterministic choice the sequences of actions corresponding to the alternative control flows. In the acyclic case, multiple input transitions are dealt with in a similar way.

To represent **cycle** constructions (Fig. 2(e)), we first of all identify a set of simple cycles and fix an ordering of their nodes. For the activity diagram in Fig. 2(e), e.g., we isolate the simple cycles $C_1=\langle S_1, S_2, S_3\rangle$ and $C_2=\langle S_2, S_3, S_4\rangle$. For each cycle two agents are specified: one to represent its main body ($Body\_C_i$, $i = 1, 2$) and the other to monitor the possible intersection with other cycles ($Cycle\_C_i$). Also, a principal agent is associated with the diagram ($Main$). It encodes the activities that are external to cycles and triggers the relevant monitor when the initial node of a cycle is met. We use three fresh names per cycle: one for the initialization of the cycle body ($inibody\_C_i$ in the example), one for the initialization of the cycle monitor (ini_C_i), and a name to signal the end of a run of the cycle (end_C_i).

Once triggered, the monitor of cycle C_i either sends an initialization signal to $Body\_C_i$, or forces the execution of the actions leading to the initial node of the other cycle and initializes the corresponding monitor. In $Body\_C_1$ for instance, these two behaviours are represented by the first and the second alternative of the top-level non-deterministic choice of the following sub-process:

$$\overline{inibody\_C_1}.\,(ini\_C_1.\,\overline{end\_C_1}.\,Cycle\_C_1 + Cycle\_C_1) + s_1.\,\overline{ini\_C_2}.\,Cycle\_C_1.$$

If the first alternative is chosen, the cycle is run once by $Body\_C_i$ and the control goes back to $Cycle\_C_i$ via a synchronization over ini_C_i. This gives raise to two further possibilities: (i) an end_C_i signal is sent to the agent $Main$; (ii) the monitor triggers yet another execution of C_i or it induces the firing of some actions and then initializes the monitor of the other cycle. For the agent $Cycle\_C_1$ the above alternatives correspond, respectively, to the left and the right branch of $(ini\_C_1.\overline{end\_C_1}.Cycle\_C_1 + Cycle\_C_1)$. The agent $Cycle\_C_2$ has a completely analogous structure.

We call **composite activities** those activities that are associated with a separate sequence or activity diagram, say S^* as in Fig. 2(f). The diagram S^* is translated as a separate process whose starting and ending points are synchronized with the rest of the translation through two special channels, ini_s^* and end_s^*. The main diagram S is encoded in such a way that, correspondingly to the translation of the composite activity S^*, the private name end_s^* is sent over the channel ini_s^*. This unblocks the process that represents the composite activity. Also, each possible control flow in such a process is let to terminate with an output action over the parameter received on ini_s^*. So, upon synchronization over end_s^*, the control goes back to the agent corresponding to the main diagram. Fig. 2(f) shows an example where an activity of the diagram S is refined into the activity diagram S^*.

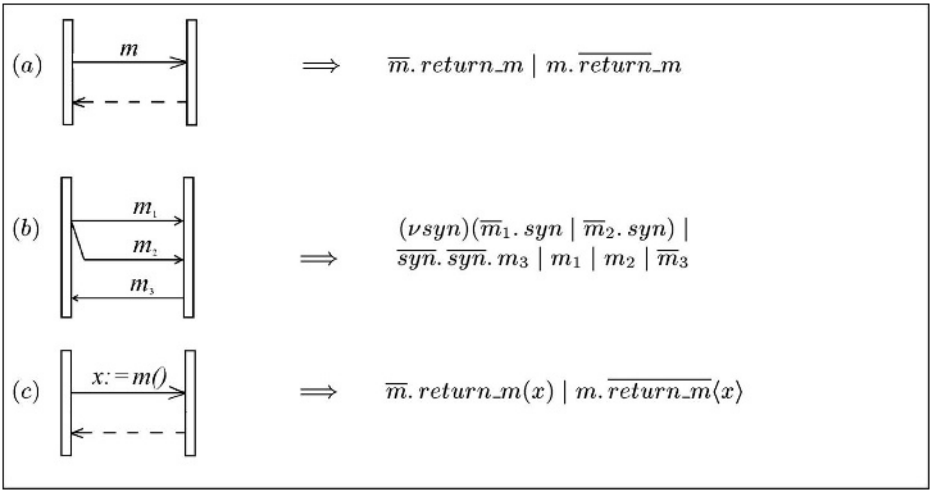

(a)
m
$\Longrightarrow$
$\overline{m}.\, return\_m \mid m.\, \overline{return\_m}$
(b)
m_1
m_2
m_3
$\Longrightarrow$
$(\nu syn)(\overline{m}_1.\, syn \mid \overline{m}_2.\, syn) \mid$
$\overline{syn}.\, \overline{syn}.\, m_3 \mid m_1 \mid m_2 \mid \overline{m}_3$
(c)
x:=m()
$\Longrightarrow$
$\overline{m}.\, return\_m(x) \mid m.\, \overline{return\_m}\langle x \rangle$

Fig. 3. Translation of elements of sequence diagrams (send and receive, explicit return, branching, assignment)

4.2 Sequence Diagrams

Relatively to sequence diagrams we exploit the translation appeared in [7]. The interested reader is referred to that paper for full details. Here we only recall a few ideas underpinning the encoding. We also comment on the scoping of the restricted names related to the adopted translation.

The objects of sequence diagrams are represented as π-calculus processes and are combined together by parallel composition. Given the diagram named D, the above parallel composition also embraces a monitor process, conventionally called SD_D, that takes care of initializing the processes translating the various objects of D.

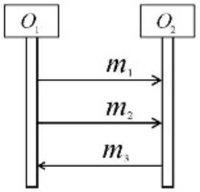
Messages between objects are rendered as communications over private channels, one per UML message. The π-process corresponding to the sending UML object performs an output action, the other process executes an input. Self-calls are interpreted as communications with additional auxiliary objects (see, e.g., the actions *encrypt* and *decrypt* and the process $OUT(encrypt, decrypt)$ in the agent $TRANS\_SearchSellerList$ below).

The explicit return of the UML message represented by the π-calculus name m is also rendered by a synchronization. The π-calculus channel adopted in this case is $return\_m$ to keep track of the name of the original message (Fig. 3(a)).

The branching of several messages is translated by making use of a fresh name for synchronization, say syn, to ensure that the receiving process does not proceed in its execution before all the messages of the branch have been delivered. In the example in Fig. 3(b), for instance, the input action m_3 is blocked by two synchronizations on syn which in turn depend on the synchronizations on both m_1 and m_2.

An assignment construction is generally used to bind the return value of a message to an identifier. The translation of assignments is then similar to that of messages with an explicit return. The single difference is that parameters of input and output action over the return channel become relevant in the translation of assignments. Hence in this case, real communication, rather than synchronization, is used (Fig. 3(c)).

The names used to encode the messages of sequence diagrams occur restricted in the global translation of the UML specification. Appropriately tuning their scope is a main issue in case one wishes to carry out performance evaluation over π-calculus terms comprising several instances of the same object. This is the case, e.g., when later on we analyze the behaviour of a system with five active sellers. As a simple example, consider the sequence diagram in Fig. 4 and imagine we want to investigate the behaviour of a system composed of one single instance of O_1 and several instances of O_2. It is necessary to grant that, whenever O_1 starts interacting with a certain instance of O_2, they keep communicating without interference from any of the other instances of O_2. To achieve that, the messages m_1, m_2, and m_3 are translated as private names sent by the monitor agent SD_D as (polyadic) parameter of the communication that initializes the objects O_1 and O_2.



$\Longrightarrow$

$$\begin{aligned} SD\_D = (\nu m_1, m_2, m_3)\, & ini\_SD\_D\langle end\_SD\_D\rangle. \\ & \overline{ini\_O_1}\langle m_1, m_2, m_3\rangle. \\ & \overline{ini\_O_2}\langle m_1, m_2, m_3\rangle. \\ & \overline{end\_SD\_D}.\, SD\_D \end{aligned}$$

$$O_1 = ini\_O_1\langle m_1, m_2, m_3\rangle.\, \overline{m_1}.\, \overline{m_2}.\, m_3.\, O_1$$
$$O_2 = ini\_O_2\langle m_1, m_2, m_3\rangle.\, m_1.\, m_2.\, \overline{m_3}.\, O_2$$

Fig. 4. Several instances of the same object: an example

4.3 Scoping

A UML specification is globally represented as the π-calculus process given by the parallel composition of the processes corresponding to the translation of the activity and the sequence diagrams in the original specification (recall that some of them might be the refinement of some activities occurring in other diagrams).

We already commented on the fact that some encodings exploit auxiliary names for synchronization and monitoring purposes, and that those names occur restricted in the translation of the global system. The table below summarizes the scope associated with those names depending on their role in the translation. In the second column of the table we use the name $TRANS\_D$ to mean the stochastic π-calculus process that translates the UML diagram D.

restricted name used as	scope
ini_action for a fork construction in diagram D	$TRANS\_D$
ini_action for a join construction in diagram D	$TRANS\_D$
ini_C, *inibody_C*, *end_C* for the cycle C in diagram D	$TRANS\_D$
ini_action for diagram D (if D is a seq. diagram this is the *ini_action* of its monitor)	global
end_action for the composite activity D* in D (if D* is a seq. diagram this is the *end_action* of its monitor)	$TRANS\_D$
ini_action for an object in the sequence diagram D	$TRANS\_D$
translation of a self-call in the sequence diagram D	$TRANS\_D$
translation of a message in the seq. diagram D (self-calls excluded)	SD_D
synchronization signal for a message with explicit return in D	$TRANS\_D$

4.4 Measures

The last step in the translation of UML specifications into processes of the stochastic π-calculus is the proper setting of the rates associated with names.

For the case study at hand, such a setting is based on the following assumptions. Any Buyer can send requests as frequently as it wants but can never produce parallel requests.

The modulation and coding scheme of the wireless physical channels in the deployment diagram (Fig. 10) is supposed to be CS-2 [5], so the data transfer rate over those channels is of 13.4 kbit/sec. From this datum, fixing the default size of request messages to be 100 bytes, we get a rate of 17 requests/sec for the communications between Seller and Buyer (see below the rates of $m3$ and $m4$ in the agent $SD\_SearchSellerList$).

Eventually, we assume that the default average time of encryption/decryption operations is 0.1 sec (then the rate of, e.g., *encrypt* in the agent $TRANS\_SearchSellerList$ is 10).

In our translation we use rates to encode probabilistic choices, too. In that case, it is sufficient to choose rates proportional to the probability of each branch in the choice construction (see in $TRANS\_ECommerce$ the rates associated with $v1$, $v2$ and with $w1$, $w2$).

4.5 Translation of the Case Study

We show below the stochastic π-calculus process that globally translates the UML specification of the web-based micro-business case study.

The translation makes use of (a family of) auxiliary agents called $OUT(\tilde{y})$ that can continuously offer output actions over their parameters. Precisely, for $N = \{n_1, \ldots, n_j\}$, the agent $OUT(N)$ is defined to be $\Sigma_{n_i \in N}\, \overline{n_i}.\, OUT(N)$.

The stochastic π-process associated with the UML specification of the case study is the following:

$$
\begin{aligned}
System = (\nu\, A,\ & ini\_ECommerce,\ ini\_PrepareBasket,\\
& ini\_SD\_SearchSellerList,\ ini\_SD\_Handshake,\ ini\_SD\_Authenticate\ )\\
& (\ TRANS\_ECommerce \mid TRANS\_SearchSellerList \mid\\
& \quad TRANS\_Authenticate \mid TRANS\_Handshake \mid\\
& \quad TRANS\_PrepareBasket \mid OUT(A,\ I)\ )
\end{aligned}
$$

where

- the agent $TRANS\_X$ is the translation of diagram X;
- A is the set of names associated with the non-composite activities from all the diagrams of the UML specification (like, e.g., $start$, $performTransaction$, ... from the E-Commerce activity diagram);
- I is the set of the ini_actions of the diagrams that are not recognized as composite activities ($ini\_ECommerce$ only in our case);

The agent $OUT(A,\ I)$ offers complementary actions to all the input actions over names in both A and I. Hence the parallel component $OUT(A,\ I)$ of $System$, together with the restriction over the names in A, closes up the specification of $System$ as it is required by the BioSpi tool. Also observe that, if i is the ini_action for a certain diagram D* and $i \notin I$ then D* corresponds to a composite activity in some other diagram D. Then the proper initialization of D* via an output action over i is carried out by the process translating D (see Fig. 2(f)).

For the sake of space, we show below the definition of two agents only: one for the translation of an activity diagram ($TRANS\_ECommerce$) and the other for the translation of a sequence diagram ($TRANS\_SearchSellerList$).

First we comment on $TRANS\_ECommerce$. Its specification reflects the features we explained for the translation of UML diagrams and the scoping of names. In particular, the following observation hold.

- The activities Handshake, Authenticate, SearchSellerList and PrepareBasket are dealt with as composite activities, and are let to correspond to the homonymous diagrams.
- Two symple cycles are identified in the E-Commerce diagram:
  C$_1$=⟨SellectSeller, Handshake⟩ and
  C$_2$=⟨SellectSeller, Handshake, Authenticate, SearchSellerList⟩.
- Relatively to the branch construction that follows the Handshake activity in the E-Commerce diagram, it is assumed that the probability to return to SelectSeller is 0.1. This is rendered by using two private names ($v1$ and $v2$) to guard the two alternatives of the choice constructor corresponding to the UML branch (see the agent $Cycle1$) and associating them the rates 9000 and 1000, respectively.

- The possible continuations after the activity SearchSellerList are supposed to be equi-probable. This is interpreted by letting the relevant choice operator of $Cycle2$ be guarded by input actions over the private names $w1$ and $w2$, both of which are associated with the rate 1000.
- To offer complementary actions to the inputs over $v1, v2, w1$ and $w2$, the ancillary agent $OUT(v1, v2, w1, w2)$ is composed in parallel with the agents directly describing the E-Commerce activity diagram.

Summing up, the definition of $TRANS\_ECommerce$ is as follows.

$$
\begin{aligned}
&TRANS\_ECommerce = \\
&(\nu\, v1[9000],\ v2[1000],\ w1[1000],\ w2[1000],\ end\_SD\_Handshake, \\
&\quad end\_SD\_Authenticate,\ end\_SD\_SearchSellerList,\ end\_PrepareBasket, \\
&\quad ini\_Cycle1,\ inibody\_Cycle1,\ end\_Cycle1,\ ini\_Cycle2,\ inibody\_Cycle2,\ end\_Cycle2) \\
&(ECommerce \mid Cycle1 \mid Body\_Cycle1 \mid Cycle2 \mid Body\_Cycle2 \mid OUT(v1, v2, w1, w2))
\end{aligned}
$$

$$
\begin{aligned}
ECommerce = {} & ini\_ECommerce. \\
& start. \\
& \overline{ini\_Cycle1}. \\
& (end\_Cycle1. \\
& \qquad \overline{ini\_SD\_Authenticate}\langle end\_SD\_Authenticate\rangle. \\
& \qquad end\_SD\_Authenticate. \\
& \qquad \overline{ini\_SD\_SearchSellerList}\langle end\_SD\_SearchSellerList\rangle. \\
& \qquad end\_SD\_SearchSellerList. \\
& \qquad \overline{ini\_PrepareBasket}\langle end\_PrepareBasket\rangle. \\
& \qquad end\_PrepareBasket. \\
& \qquad performTransaction. \\
& \qquad closeSession. \\
& \qquad ECommerce) \\
& + \\
& (end\_Cycle2. \\
& \qquad \overline{ini\_PrepareBasket}\langle end\_PrepareBasket\rangle. \\
& \qquad end\_PrepareBasket. \\
& \qquad performTransaction. \\
& \qquad closeSession. \\
& \qquad ECommerce)
\end{aligned}
$$

$$
\begin{aligned}
Cycle1 = {} & ini\_Cycle1. \\
& (\overline{inibody\_Cycle1}.\, (v1.\, ini\_Cycle1.\, \overline{end\_Cycle1}.\, Cycle1 + v2.\, Cycle1) \\
& \ +\ \overline{ini\_Cycle2}.\, Cycle1)
\end{aligned}
$$

$$
\begin{aligned}
Body\_Cycle1 = {} & inibody\_Cycle1. \\
& selectSeller. \\
& \overline{ini\_SD\_Handshake}\langle end\_SD\_Handshake\rangle. \\
& end\_SD\_Handshake. \\
& \overline{ini\_Cycle1}. \\
& Body\_Cycle1
\end{aligned}
$$

$$\begin{aligned}
\mathit{Cycle}2 = {} & \mathit{ini\_Cycle}2.\\
& (\overline{\mathit{inibody\_Cycle}2}.\,(w1.\,\mathit{ini\_Cycle}2.\,\overline{\mathit{end\_Cycle}2}.\,\mathit{Cycle}2 + w2.\,\mathit{Cycle}2)\\
& \;+\; \overline{\mathit{ini\_Cycle}1}.\,\mathit{Cycle}2)
\end{aligned}$$

$$\begin{aligned}
\mathit{Body\_Cycle}2 = {} & \mathit{inibody\_Cycle}2.\\
& \mathit{selectSeller}.\\
& \overline{\mathit{ini\_SD\_Handshake}}\langle \mathit{end\_SD\_Handshake}\rangle.\\
& \mathit{end\_SD\_Handshake}.\\
& \overline{\mathit{ini\_SD\_Authenticate}}\langle \mathit{end\_SD\_Authenticate}\rangle.\\
& \mathit{end\_SD\_Authenticate}.\\
& \overline{\mathit{ini\_SD\_SearchSellerList}}\langle \mathit{end\_SD\_SearchSellerList}\rangle.\\
& \overline{\mathit{end\_SD\_SearchSellerList}}.\\
& \overline{\mathit{ini\_Cycle}2}.\\
& \mathit{Body\_Cycle}2
\end{aligned}$$

We report below the translation of the SearchSellerList sequence diagram. Only two issues are worth mentioning about it.

- The input actions over the private names *encrypt* and *decrypt* in the agents $\mathit{SM}1\_\mathit{SearchSellerList}$ and $\mathit{SM}2\_\mathit{SearchSellerList}$ are used to translate the self-calls in the corresponding objects of the UML diagram. Also, complementary actions are granted by the auxiliary parallel agent $\mathit{OUT}(\mathit{encrypt},\ \mathit{decrypt})$.
- The restricted name $\mathit{end\_DS}$ is due to render the explicit return of the message sent by the object DS of the SearchSellerList diagram. After such a return message has been delivered, the monitor $\mathit{SD\_SearchSellerList}$ of the sequence diagram can pass the control back to the process $\mathit{ECommerce}$ via an output action over its $\mathit{end\_action}$, namely $\mathit{end\_SD\_SearchSellerList}$.

$$\begin{aligned}
& \mathit{TRANS\_SearchSellerList} =\\
& (\nu\, \mathit{encrypt}[10],\ \mathit{decrypt}[10],\ \mathit{ini\_DS},\ \mathit{ini\_SM}1,\ \mathit{ini\_SM}2,\ \mathit{ini\_LM},\ \mathit{end\_DS}\ )\\
& (\ \mathit{SD\_SearchSellerList} \mid \mathit{DS\_SearchSellerList} \mid \mathit{SM}1\_\mathit{SearchSellerList} \mid\\
& \;\; \mathit{SM}2\_\mathit{SearchSellerList} \mid \mathit{LM\_SearchSellerList} \mid \mathit{OUT}(\mathit{encrypt},\ \mathit{decrypt})\ )
\end{aligned}$$

$$\begin{aligned}
\mathit{SD\_SearchSellerList} = {} & (\nu\ m1,\ m2,\ m3[17],\ m4[17],\ m5,\ m6)\\
& \mathit{ini\_SD\_SearchSellerList}(\mathit{end\_SD\_SearchSellerList}).\\
& \overline{\mathit{ini\_DS}}\langle m1, m2\rangle.\\
& \overline{\mathit{ini\_SM}1}\langle m1, m2, m3, m4\rangle.\\
& \overline{\mathit{ini\_SM}2}\langle m3, m4, m5, m6\rangle.\\
& \overline{\mathit{ini\_LM}}\langle m5, m6\rangle.\\
& \overline{\mathit{end\_DS}}.\\
& \overline{\mathit{end\_SD\_SearchSellerList}}.\\
& \mathit{SD\_SearchSellerList}
\end{aligned}$$

$$\begin{aligned}
\mathit{DS\_SearchSellerList} = {} & \\
& \mathit{ini\_DS}(\mathit{putMes}, \mathit{returnPutMes}).\\
& \overline{\mathit{putMes}}\langle \mathit{seller}, \mathit{searchItem}\rangle.\\
& \mathit{returnPutMes}(\mathit{info}).\\
& \mathit{end\_DS}.\\
& \mathit{DS\_SearchSellerList}
\end{aligned}$$

$$\begin{aligned}
&SM1\_SearchSellerList = \\
&\qquad ini\_SM1(putMes, returnPutMes, secureSend\_1, secureSend\_2). \\
&\qquad putMes(seller, searchItem). \\
&\qquad encrypt. \\
&\qquad \overline{secureSend\_1}\langle encData, seller\rangle. \\
&\qquad secureSend\_2(encData, buyer). \\
&\qquad decrypt. \\
&\qquad \overline{returnPutMes}\langle info\rangle. \\
&\qquad SM1\_SearchSellerList
\end{aligned}$$

$$\begin{aligned}
&SM2\_SearchSellerList = \\
&\qquad ini\_SM2(secureSend\_1, secureSend\_2, searchList, putMes). \\
&\qquad secureSend\_1(encData, seller). \\
&\qquad decrypt. \\
&\qquad \overline{searchList}\langle data\rangle. \\
&\qquad putMes(buyer, result). \\
&\qquad encrypt. \\
&\qquad \overline{secureSend\_2}\langle encData, buyer\rangle. \\
&\qquad SM2\_SearchSellerList
\end{aligned}$$

$$\begin{aligned}
&LM\_SearchSellerList = \\
&\qquad ini\_LM(searchList, putMes). \\
&\qquad searchList(data). \\
&\qquad \overline{putMes}\langle buyer, result\rangle. \\
&\qquad LM\_SearchSellerList
\end{aligned}$$

5 Performance Analysis

The stochastic π-calculus specification of the web-based micro-business case study was simulated using the BioSPI tool, an application based on the Logix system, which implements Flat Concurrent Prolog (FCP) [10, 11]. We conclude the paper by reporting on the performance evaluation that was carried out. In particular, here we focus on three main issues: the time to serve concurrent requests, the time to authenticate the buyer, and the complexity of the encryption/decryption algorithms.

The BioSpi tool allows the user to maintain a full record of the evolution of each process in the system. The record specifies all the communications the process has been involved in, the time and channel at which they occurred, the communicating partner, and the processes resulting from each communication. Moreover, in BioSpi the user can dynamically set the number of instances of processes. This is an essential feature when one is interested in comparing slightly different configurations of the same system, with a distinct number of components at a time.

The plots in Fig. 5 represent the time to serve concurrent requests in two distinct cases: when one single Seller (1-Seller) is available (upper curve), and

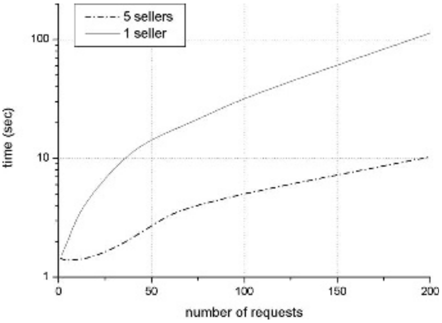


Fig. 5. Time to serve concurrent requests

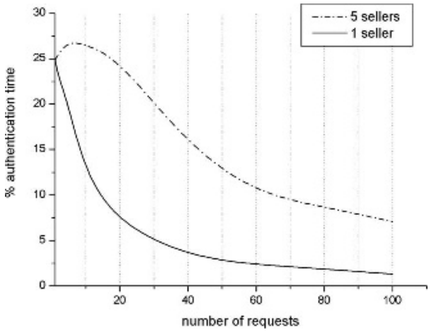


Fig. 6. Authentication time

when five Sellers (5-Sellers) are available (lower curve). As expected, the global system shows to be very sensitive to the number of available Sellers. Fixing a certain response time we can derive the maximal number of Buyers which are simultaneously allowed to access the system. For example, if the response time is set to 10, then 35 Buyers can be allowed in the 1-Seller system, and 200 in the 5-Sellers system, respectively.

The authentication time depends on both the number of parallel requests and on the complexity of the cryptographic algorithms used for secure transmission of data. The curves in Fig. 6 show the dependency of the authentication time on the number of concurrent requests. For example, they make clear that the time spent in authentication operations is negligible in the 1-Seller system when the number of requests is greater than 50.

The plots in Fig. 7 show how the complexity of the cryptographic algorithms can influence the relative authentication time, that in turn is obtained as a ratio between the absolute authentication time and the time needed to serve concurrent requests. The curve for 5-Sellers reveals, e.g., that if the encryption algorithm takes more than 0.5 sec to run then it is the most influent factor for

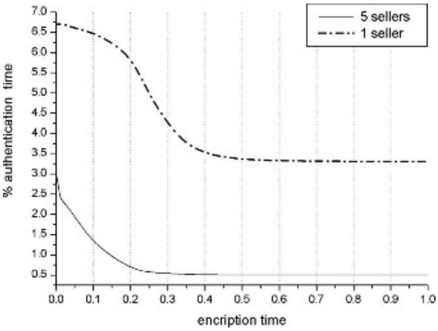


Fig. 7. Influence of cryptographic algorithms over authentication time

the time needed to serve concurrent requests and hence for the throughput of the whole system.

Acknowledgements. We are grateful to Paola Lecca for her useful comments on the subject, and to William Silverman for his support on the use of the BioSpi tool.

References

1. BioSpi home page: `http://www.wisdom.weizmann.ac.il/~biopsi/`.
2. Booch, G., Rumbaugh, J. and Jacobson, I.: The Unified Modeling Language User Guide, Addison-Wesley (1998).
3. Caraguili C., Piazza D., Mura I. et al.: Specification in UML of case studies. DEGAS project deliverable 24, `http://www.omnys.it/degas/` (2002).
4. DEGAS home page: `http://www.omnys.it/degas/`.
5. Kalden R., Mierick I., Meyer M.: Wireless Internet Access Based on GPRS. IEEE Personal Comm. 7, 8-18 (2000).
6. Milner R., *Communicating and Mobile Systems: the π-calculus*, Cambridge University Press (1999).
7. Pokozy-Korenblat K., Priami C.: Toward extracting pi-calculus from UML sequence and state diagrams. Proceedings of the workshop on Compositional verification of UML models'03, ENTCS, to appear (2003).
8. Priami C., Regev A., Silverman W. and Shapiro E.: Application of a stochastic name passing calculus to representation and simulation of molecular processes. Information Processing Letters 80: 25-31 (2001).
9. D. Sangiorgi and D. Walker. **The π-calculus: a Theory of Mobile Processes**. Cambridge University Press (2001).
10. Shapiro E.: Concurrent prolog: a progress report. In Shapiro E., editor, *Concurrent Prolog (vol. 1)*, pages 157 - 187. MIT Press, Cambridge, Massachusetts (1987).
11. Silverman W., Hirsh M., Houri A. and Shapiro E., *The Logix system user manual, Version 1.21 - Concurrent Prolog (vol. II).* MIT Press (1987).

Appendix: UML Specification of the Case Study

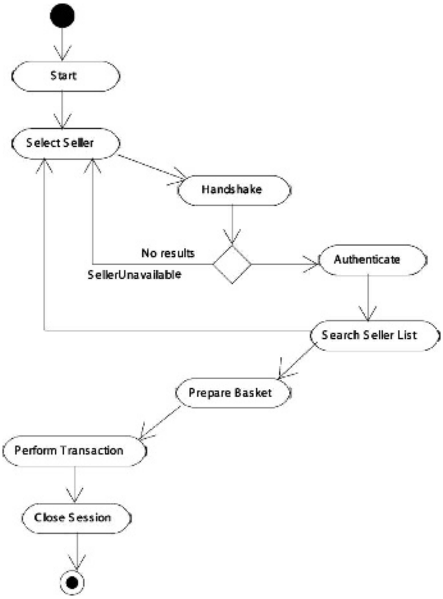


Fig. 8. E-Commerce activity diagram

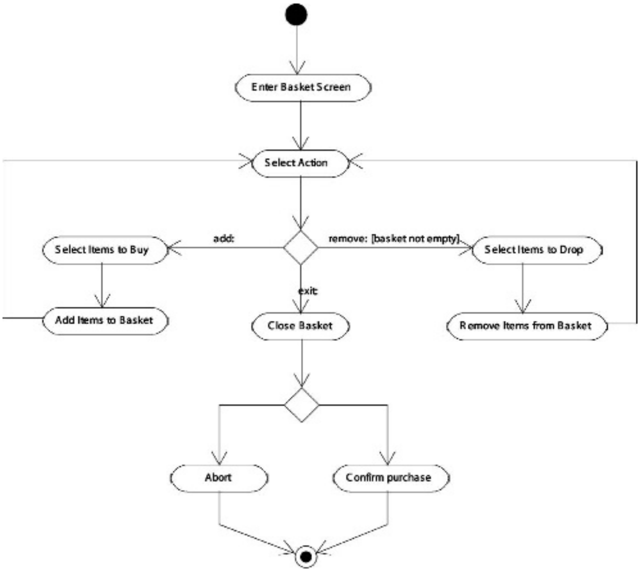


Fig. 9. PrepareBasket activity diagram

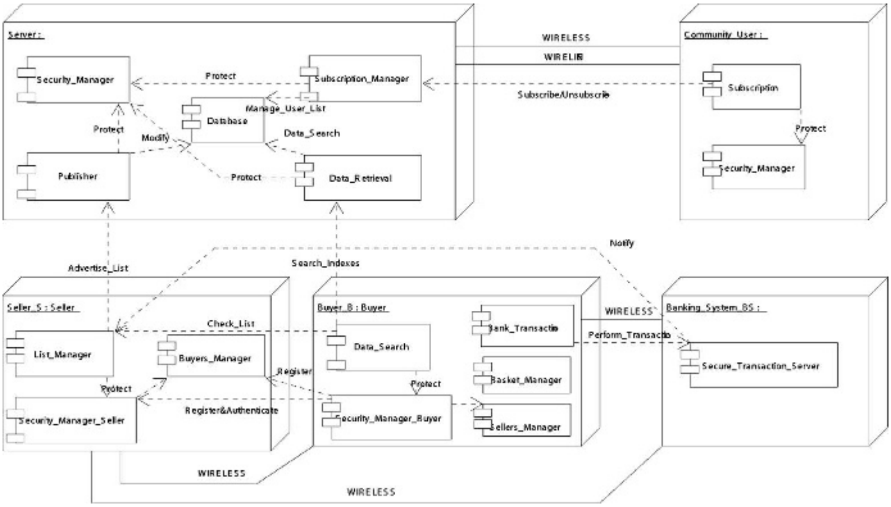


Fig. 10. Deployment diagram

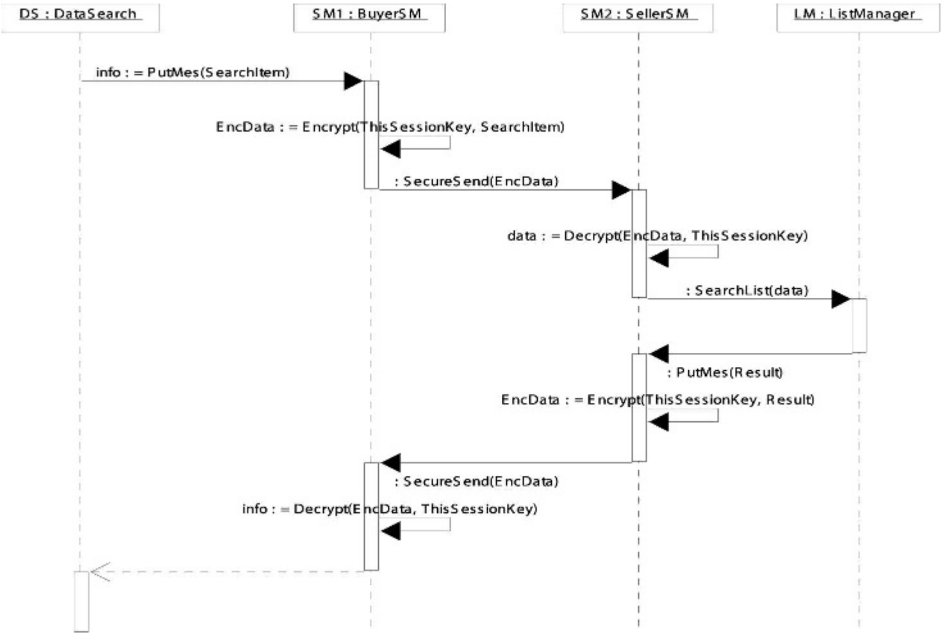

DS : DataSearch
SM1 : BuyerSM
SM2 : SellerSM
LM : ListManager
info : = PutMes(SearchItem)
EncData : = Encrypt(ThisSessionKey, SearchItem)
: SecureSend(EncData)
data : = Decrypt(EncData, ThisSessionKey)
: SearchList(data)
: PutMes(Result)
EncData : = Encrypt(ThisSessionKey, Result)
: SecureSend(EncData)
info : = Decrypt(EncData, ThisSessionKey)

Fig. 11. SearchSellerList sequence diagram

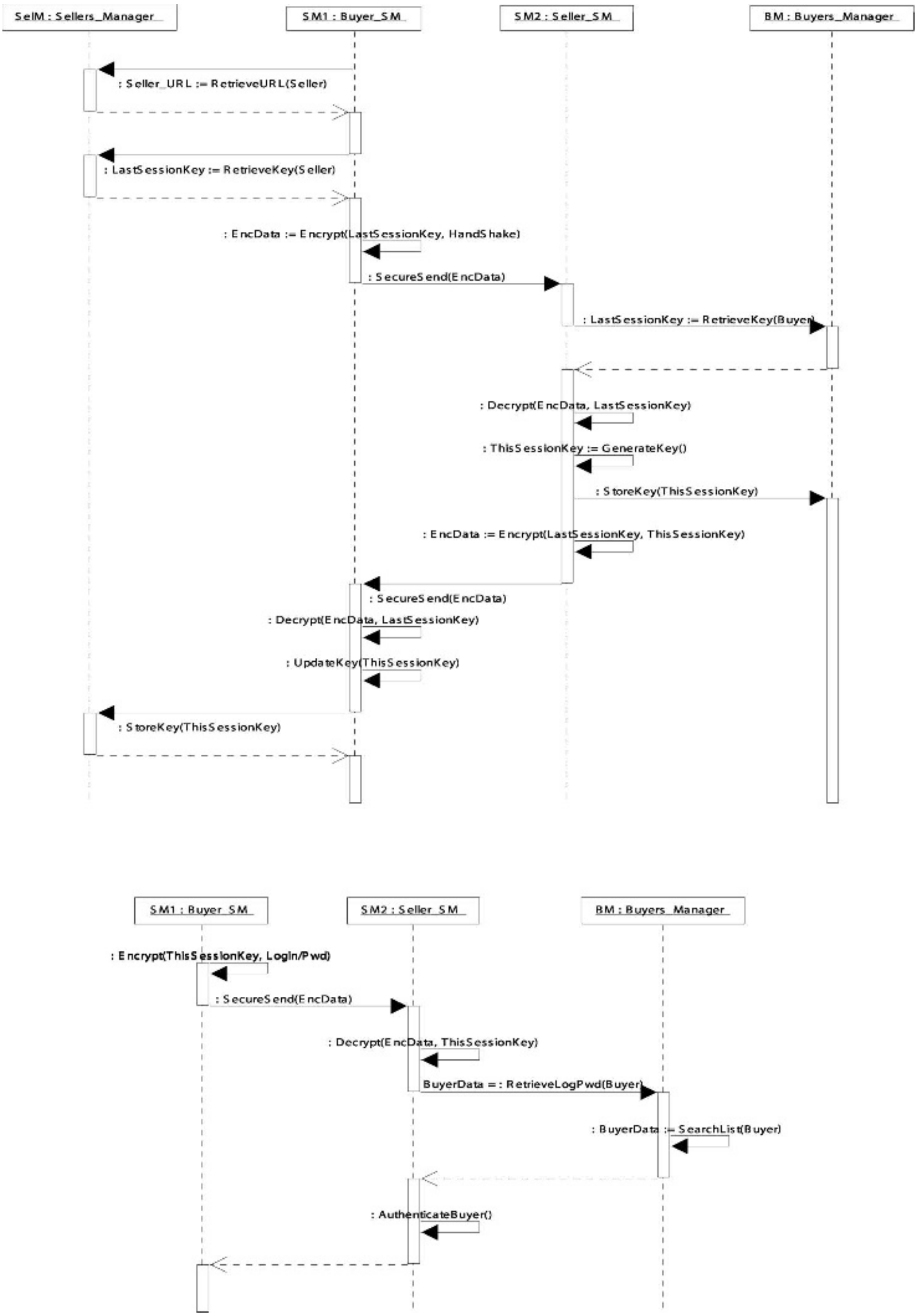

SelM : Sellers_Manager
SM1 : Buyer_SM
SM2 : Seller_SM
BM : Buyers_Manager
: Seller_URL := RetrieveURL(Seller)
: LastSessionKey := RetrieveKey(Seller)
: EncData := Encrypt(LastSessionKey, HandShake)
: SecureSend(EncData)
: LastSessionKey := RetrieveKey(Buyer)
: Decrypt(EncData, LastSessionKey)
: ThisSessionKey := GenerateKey()
: StoreKey(ThisSessionKey)
: EncData := Encrypt(LastSessionKey, ThisSessionKey)
: SecureSend(EncData)
: Decrypt(EncData, LastSessionKey)
: UpdateKey(ThisSessionKey)
: StoreKey(ThisSessionKey)
SM1 : Buyer_SM
SM2 : Seller_SM
BM : Buyers_Manager
: Encrypt(ThisSessionKey, Login/Pwd)
: SecureSend(EncData)
: Decrypt(EncData, ThisSessionKey)
BuyerData =: RetrieveLogPwd(Buyer)
: BuyerData := SearchList(Buyer)
: AuthenticateBuyer()

Fig. 12. Handshake (top) and Authenticate (bottom) sequence diagrams

Efficient Information Propagation Algorithms in Smart Dust and NanoPeer Networks[*]

Sotiris Nikoletseas and Paul Spirakis

Department of Computer Engineering and Informatics,
University of Patras and Computer Technology Institute (CTI), Greece
{nikole, spirakis}@cti.gr

Abstract. Wireless sensor networks are comprised of a vast number of ultra-small, fully autonomous computing, communication and sensing devices, with very restricted energy and computing capabilities, that co-operate to accomplish a large sensing task. The efficient and robust realization of such large, highly-dynamic and complex networking environments is a challenging algorithmic and technological task.

In this work we present and discuss two protocols for efficient and robust data propagation in wireless sensor networks: LTP (a "local target" optimization protocol) and PFR (a multi-path probabilistic forwarding protocol). Furthermore, we present the NanoPeers architecture paradigm, a peer-to-peer network of lightweight devices, lacking all or most of the capabilities of their computer-world counterparts. We identify the problems arising when applying current routing and searching methods to this nano-world, and present some initial solutions, using a case study of a sensor network instance; Smart Dust.

1 Introduction

1.1 A Description of Wireless Sensor Networks

Recent dramatic developments in micro-electro-mechanical (MEMS) systems, wireless communications and digital electronics have already led to the development of small in size, low-power, low-cost sensor devices. Such extremely small devices integrate sensing, data processing and wireless communication capabilities. Current devices have a size at the cubic centimeter scale, a CPU running at 4 MHz, some memory and a wireless communication capability at a 4kbps rate. Also, they are equipped with a small but effective operating system and are able to switch between "sleeping" and "awake" modes to save energy. Pioneering groups (like the "Smart Dust" Project at Berkeley, the "Wireless Integrated Network Sensors" Project at UCLA and the "Ultra low Wireless Sensor" Project at

[*] This work has been partially supported by the IST/FET Programme of the European Union under contract number IST-2001-33116 (FLAGS) and within the 6FP under contract 001907 (DELIS).

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 127–145, 2004.

MIT) pursue further important goals, like a total volume of a few cubic millimeters and extremely low energy consumption, by using alternative technologies, based on radio frequency (RF) or optical (laser) transmission.

Examining each such device individually might appear to have small utility, however the effective *distributed co-ordination* of large numbers of such devices may lead to the efficient accomplishment of large sensing tasks. Large numbers of sensor nodes can be deployed in areas of interest (such as inaccessible terrains or disaster places) and use *self-organization and collaborative methods* to form a sensor network.

Their wide range of applications is based on the possible use of various sensor types (i.e. thermal, visual, seismic, acoustic, radar, magnetic, etc.) in order to monitor a wide variety of conditions (e.g. temperature, object presence and movement, humidity, pressure, noise levels etc.). Thus, sensor networks can be used for continuous sensing, event detection, location sensing as well as micro-sensing. Hence, sensor networks have important applications, including (a) military (like forces and equipment monitoring, battlefield surveillance, targeting, nuclear, biological and chemical attack detection), (b) environmental applications (such as fire detection, flood detection, precision agriculture), (c) health applications (like telemonitoring of human physiological data) and (d) home applications (e.g. smart environments and home automation). For an excellent survey of wireless sensor networks see [2] and also [10, 13].

1.2 Critical Challenges

The efficient and robust realization of such large, highly-dynamic, complex, non-conventional networking environments is *a challenging algorithmic and technological task*. Features including the huge number of sensor devices involved, the severe power, computational and memory limitations, their dense deployment and frequent failures, pose *new design, analysis and implementation aspects* which are essentially different not only with respect to distributed computing and systems approaches but also to ad-hoc networking techniques.

We emphasize the following characteristic differences between sensor networks and ad-hoc networks:

- The number of sensor particles in a sensor network is extremely large compared to that in a typical ad-hoc network.
- Sensor networks are typically prone to faults.
- Because of faults as well as energy limitations, sensor nodes may (permanently or temporarily) join or leave the network. This leads to highly dynamic network topology changes.
- The density of deployed devices in sensor networks is much higher than in ad-hoc networks.
- The limitations in energy, computational power and memory are much more severe in sensor networks.

Because of the above rather unique characteristics of sensor networks, efficient and robust distributed protocols and algorithms should exhibit the following critical properties:

Scalability

Distributed protocols for sensor networks should be highly scalable, in the sense that they should operate efficiently in extremely large networks composed of huge numbers of nodes. This feature calls for an urgent need to prove by analytical means and also validate (by large scale simulations) certain efficiency and robustness (and their trade-offs) guarantees for asymptotic network sizes.

Efficiency

Because of the severe energy limitations of sensor networks and also because of their time-critical application scenaria, protocols for sensor networks should be efficient, with respect to both energy and time.

Fault-Tolerance

Sensor particles are prone to several types of faults and unavailabilities, and may become inoperative (permanently or temporarily). Various reasons for such faults include physical damage during either the deployment or the operation phase, permanent (or temporary) cease of operation in the case of power exhaustion (or energy saving schemes, respectively). The sensor network should be able to continue its proper operation for as long as possible despite the fact that certain nodes in it may fail.

For a further discussion on challenges and future problems in sensor nets research, the reader may see [5, 6, 20].

1.3 Overview of This Work

We present, analyse and discuss two energy and time efficient protocols:

- *The Local Target Protocol (LTP)*, that performs a local optimization trying to minimize the number of data transmissions needed for the information to reach the sink.
- *The Probabilistic Forwarding Protocol (PFR)*, that creates redundant data transmissions that are probabilistically optimized, to trade-off energy efficiency with fault-tolerance.

Furthermore, we present the NanoPeers architecture paradigm, a peer-to-peer network of lightweight devices, lacking all or most of the capabilities of their computer-world counterparts. We identify the problems arising when we apply current routing and searching methods to this nano-world, and present some initial solutions, using a case study of a sensor network instance; Smart Dust. Our position is that (i) experience gained through research and experimentation in the field of P2P computing can be indispensable in environments of more restricted computing capabilities, and that (ii) the proposed framework can be the basis of numerous real-world applications, opening up several challenging research problems.

2 LTP: A Hop-by-Hop Data Propagation Protocol

2.1 The Model

We adopt a two-dimensional (plane) framework: A *smart dust cloud* (a set of particles) is spread in an area (for a graphical presentation, see Fig. 1).

Let d (usually measured in numbers of $particles/m^2$) be the *density* of particles in the area. Let $\mathcal{R}$ be the maximum (radio/laser) transmission range of each grain particle.

A *receiving wall* $\mathcal{W}$ is defined to be an infinite line in the smart-dust plane. Any particle transmission within range $\mathcal{R}$ from the wall $\mathcal{W}$ is received by $\mathcal{W}$. $\mathcal{W}$ is assumed to have very strong computing power, able to collect and analyze received data and has a constant power supply and so it has no energy constraints. The wall represents in fact the authorities (the fixed control center) who the realization of a crucial event should be reported to. The wall notion generalizes that of the sink and may correspond to multiple (and/or moving) sinks. Note that a wall of appropriately big (finite) length suffices.

Furthermore, there is a set-up phase of the smart dust network, during which the smart cloud is dropped in the terrain of interest, when using special control messages (which are very short, cheap and transmitted only once) each smart dust particle is provided with the direction of $\mathcal{W}$. By assuming that each smart-dust particle has individually *a sense of direction*, and using these control messages, each particle is aware of the general location of $\mathcal{W}$.

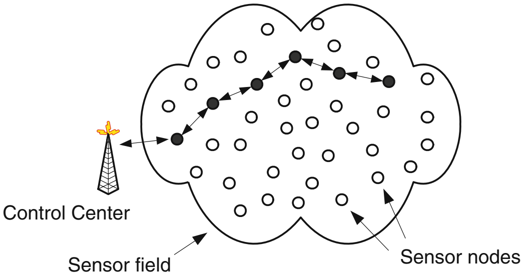


Fig. 1. A Smart Dust Cloud

2.2 The Protocol

Let $d(p_i, p_j)$ the distance (along the corresponding vertical lines towards $\mathcal{W}$) of particles p_i, p_j and $d(p_i, \mathcal{W})$ the (vertical) distance of p_i from $\mathcal{W}$. Let *info(*$\mathcal{E}$*)* the information about the realization of the crucial event $\mathcal{E}$ to be propagated. Let p the particle sensing the event and starting the execution of the protocol. In this protocol, each particle p' that has received *info(*$\mathcal{E}$*)*, does the following:

- *Search Phase*: It uses a periodic low energy directional broadcast in order to discover a particle nearer to $\mathcal{W}$ than itself. (i.e. a particle p'' where $d(p'', \mathcal{W}) < d(p', \mathcal{W})$).
- *Direct Transmission Phase*: Then, p' sends *info($\mathcal{E}$)* to p''.
- *Backtrack Phase*: If consecutive repetitions of the *search phase* fail to discover a particle nearer to $\mathcal{W}$, then p' sends *info($\mathcal{E}$)* to the particle that it originally received the information from.

Note that one can estimate an a-priori upper bound on the number of repetitions of the search phase needed, by calculating the probability of success of each search phase, as a function of various parameters (such as density, search angle, transmission range). This bound can be used to decide when to backtrack.

For a graphical representation see Fig. 2, Fig. 3. The LTP protocol was introduced in [9].

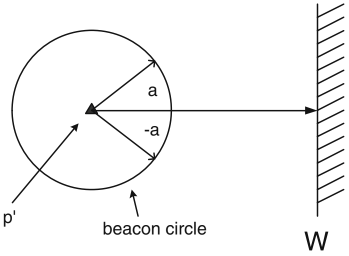


Fig. 2. Example of the Search Phase

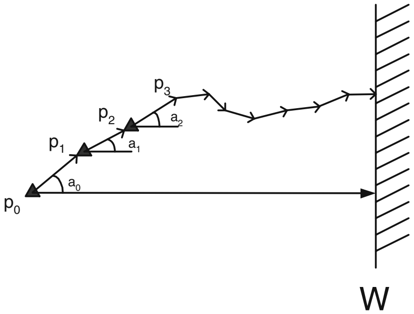


Fig. 3. Example of a Transmission

2.3 Analysis

We first provide some basic definitions.

Definition 1. Let $h_{opt}(p, \mathcal{W})$ be the (optimal) number of "hops" (direct, vertical to $\mathcal{W}$ transmissions) needed to reach the wall, in the *ideal* case in which particles always exist in pair-wise distances $\mathcal{R}$ on the vertical line from p to $\mathcal{W}$. Let Π be *a smart-dust propagation protocol*, using *a transmission path* of length $L(\Pi, p, \mathcal{W})$ to send info about event $\mathcal{E}$ to wall $\mathcal{W}$. Let $h(\Pi, p, \mathcal{W})$ be the actual number of hops (transmissions) taken to reach $\mathcal{W}$. The *"hops" efficiency* of protocol Π is the ratio

$$C_h = \frac{h(\Pi, p, \mathcal{W})}{h_{opt}(p, \mathcal{W})}$$

Clearly, the number of hops (transmissions) needed characterizes the energy consumption and the time needed to propagate the information $\mathcal{E}$ to the wall. Remark that $h_{opt} = \left\lceil \frac{d(p,\mathcal{W})}{\mathcal{R}} \right\rceil$, where $d(p, \mathcal{W})$ is the (vertical) distance of p from the wall $\mathcal{W}$.

In the case where the protocol Π is randomized, or in the case where the distribution of the particles in the cloud is a random distribution, the number of hops h and the efficiency ratio C_h are random variables and one wishes to study their expected values.

The reason behind these definitions is that when p (or any intermediate particle in the information propagation to $\mathcal{W}$) "looks around" for a particle as near to $\mathcal{W}$ as possible to pass its information about $\mathcal{E}$, it may not get any particle in the perfect direction of the line vertical to $\mathcal{W}$. This difficulty comes mainly from three causes: a) Due to the initial spreading of particles of the cloud in the area and because particles do not move, there might not be any particle in that direction. b) Particles of sufficient remaining battery power may not be currently available in the right direction. c) Particles may temporarily "sleep" (i.e. not listen to transmissions) in order to save battery power.

Note that any given distribution of particles in the smart dust cloud may not allow the ideal optimal number of hops to be achieved at all. In fact, the least possible number of hops depends on the input (the positions of the grain particles). We however, compare the efficiency of protocols to the ideal case. A comparison with the best achievable number of hops in each input case will of course give better efficiency ratios for protocols.

To enable a first step towards a rigorous analysis of smart dust protocols, we make the following simplifying assumption: *The search phase always finds a* p'' (of sufficiently high battery) in the semicircle of center the particle p' currently possessing the information about the event and radius R, in the direction towards $\mathcal{W}$. Note that this assumption on always finding a particle can be relaxed in the following ways: (a) by repetitions of the search phase until a particle is found. This makes sense if at least one particle exists but was sleeping during the failed searches, (b) by considering, instead of just the semicircle, a cyclic sector defined by circles of radiuses $\mathcal{R} - \Delta\mathcal{R}$, $\mathcal{R}$ and also take into account the density of the smart cloud, (c) if the protocol during a search phase ultimately fails to find a particle towards the wall, it may *backtrack*.

We also assume that the position of p'' is uniform in the arc of angle 2α around the direct line from p' vertical to $\mathcal{W}$. Each data transmission (one hop) takes constant time t (so the "hops" and time efficiency of our protocols coincide in this case). It is also assumed that each target selection is stochastically *independent* of the others, in the sense that it is always drawn uniformly randomly in the arc $(-\alpha,\, \alpha)$. We call this the "α-uniform" case.

The above assumptions may not be very realistic in practice, however, they can be relaxed and in any case allow to perform a first effort towards providing some concrete analytical results.

Lemma 1. *The expected "hops efficiency" of the local target protocol in the α-uniform case is*

$$E(C_h) \;\simeq\; \frac{\alpha}{\sin\alpha}$$

for large h_{opt}. Also

$$1 \;\leq\; E(C_h) \;\leq\; \frac{\pi}{2} \simeq 1.57$$

for $0 \;\leq\; \alpha \;\leq\; \frac{\pi}{2}$.

Proof. Due to the protocol, a sequence of points is generated, $p_0 = p,\, p_1,\, p_2, \ldots,$ $p_{h-1},\, p_h$ where p_{h-1} is a particle within $\mathcal{W}$'s range and p_h is part of the wall. Let α_i be the (positive or negative) angle of p_i with respect to p_{i-1}'s vertical line to $\mathcal{W}$. It is:

$$\sum_{i=1}^{h-1} d(p_{i-1},\, p_i) \;\leq\; d(p,\, \mathcal{W}) \;\leq\; \sum_{i=1}^{h} d(p_{i-1},\, p_i)$$

Since the (vertical) progress towards $\mathcal{W}$ is then $\Delta_i \;=\; d(p_{i-1}, p_i) \;=\; \mathcal{R}\cos\alpha_i$, we get:

$$\sum_{i=1}^{h-1} \cos\alpha_i \;\leq\; h_{opt} \;\leq\; \sum_{i=1}^{h} \cos\alpha_i$$

From Wald's equation for the expectation of a sum of a random number of independent random variables (see [18]), then

$$E(h-1) \,\cdot\, E(\cos\alpha_i) \;\leq\; E(h_{opt}) \;=\; h_{opt} \;\leq\; E(h) \,\cdot\, E(\cos\alpha_i)$$

Now, $\forall i,\, E(\cos\alpha_i) \;=\; \int_{-\alpha}^{\alpha} \cos x \,\frac{1}{2\alpha} dx \;=\; \frac{\sin\alpha}{\alpha}$. Thus

$$\frac{\alpha}{\sin\alpha} \;\leq\; \frac{E(h)}{h_{opt}} \;=\; E(C_h) \;\leq\; \frac{\alpha}{\sin\alpha} + \frac{1}{h_{opt}}$$

Assuming large values for h_{opt} (i.e. events happening far away from the wall, which is the most interesting case in practice since the detection and propagation difficulty increases with distance) we have (since for $0 \;\leq\; \alpha \;\leq\; \frac{\pi}{2}$ it is $1 \;\leq\; \frac{\alpha}{\sin\alpha} \;\leq\; \frac{\pi}{2}$) and the result follows.

2.4 Local Optimization: The Min-two Uniform Targets Protocol (M2TP)

We further assume that the search phase always returns *two points* p'', p''' each uniform in $(-\alpha,\, \alpha)$ and that a modified protocol M2TP selects the best of the two points, with respect to the local (vertical) progress. This is in fact an optimized version of the Local Target Protocol.

In a similar way as in the proof of the previous lemma, we prove the following result:

Lemma 2. *The expected "hops efficiency" of the "min two uniform targets" protocol in the α-uniform case is*

$$E(C_h) \simeq \frac{\alpha^2}{2(1-\cos\alpha)}$$

for $0 \le \alpha \le \frac{\pi}{2}$ *and for large* h*.*

Now remark that

$$\lim_{\alpha\to 0} E(C_h) = \lim_{\alpha\to 0}\frac{2\alpha}{2\sin a} = 1$$

and

$$\lim_{\alpha\to\frac{\pi}{2}} E(C_h) = \frac{(\pi/2)^2}{2(1-0)} = \frac{\pi^2}{8} \simeq 1.24$$

Thus, we prove the following:

Lemma 3. *The expected "hops" efficiency of the min-two uniform targets protocol is*

$$1 \le E(C_h) \le \frac{\pi^2}{8} \simeq 1.24$$

for large h *and for* $0 \le \alpha \le \frac{\pi}{2}$*.*

Remark that, with respect to the expected hops efficiency of the local target protocol, the min-two uniform targets protocol achieves, because of the one additional search, a relative gain which is $(\pi/2 - \pi^2/8)/(\pi/2) \simeq 21.5\%$.

3 PFR: A Probabilistic Multi-path Forwarding Protocol

The LTP protocol, as shown in the previous section manages to be very efficient by always selecting exactly one next-hop particle, with respect to some optimization criterion. Thus, it tries to minimize the number of data transmissions. LTP is indeed very successful in the case of dense and robust networks, since in such networks a next hop particle is very likely to be discovered. In sparse or faulty networks however, the LTP protocol may behave poorly, because of many backtracks due to frequent failure to find a next hop particle. To combine energy efficiency and fault-tolerance, the Probabilistic Forwarding Protocol (PFR) has been introduced.

3.1 The Model

We assume the case where particles are *randomly deployed* in a given area of interest. Such a placement may occur e.g. when throwing sensors from an airplane over an area.

As a special case, we consider the network being a lattice (or grid) deployment of sensors. This grid placement of grain particles is motivated by certain applications, where it is possible to have a pre-deployed sensor network, where sensors are put (possibly by a human or a robot) in a way that they form a *2-dimensional lattice*. Note indeed that such sensor networks, deployed in a structured way, might be useful in precise agriculture, where humans or robots may want to deploy the sensors in a lattice structure to monitor in a rather homogenous and uniform way certain conditions in the spatial area of interest. Certainly, exact terrain monitoring in military applications may also need some sort of a grid-like shaped sensor network. Note also that Akyildiz et al in a recent state of the art survey ([2]) do not exclude the pre-deployment possibility. Also, [11] explicitly refers to the lattice case. Moreover, as the authors of [11] state in an extended version of their work ([12]), they consider, for reasons of “analytic tractability”, a square grid topology.

Let N be the number of deployed grain particles. There is a single point in the network area, which we call the sink S, and represents a control center where data should be propagated to.

We assume that each grain particle has the following abilities:

(i) It can estimate the direction of a received transmission (e.g. via the technology of direction-sensing antennae).

(ii) It can estimate the distance from a nearby particle that did the transmission (e.g. via estimation of the attenuation of the received signal).

(iii) It knows the direction towards the sink S. This can be implemented during a set-up phase, where the (very powerful in energy) sink broadcasts the information about itself to all particles.

(iv) All particles have a common co-ordinates system.

Notice that GPS information is not needed for this protocol. Also, there is no need to know the global structure of the network.

3.2 The Protocol

The PFR protocol is inspired by the probabilistic multi-path design choice for the Directed Diffusion paradigm mentioned in [11]. The PFR protocol was first proposed in [7]. Its basic idea of the protocol (introduced in [7]) lies in probabilistically favoring transmissions towards the sink within a *thin zone* of particles around the line connecting the particle sensing the event $\mathcal{E}$ and the sink (see Fig. 4). Note that transmission along this line is energy optimal. However it is not always possible to achieve this optimality, basically because certain sensors on this direct line might be inactive, either permanently (because their energy has been exhausted) or temporarily (because these sensors might enter a sleeping mode to save energy). Further reasons include (a) physical damage of sensors,

(b) deliberate removal of some of them (possibly by an adversary in military applications), (c) changes in the position of the sensors due to a variety of reasons (weather conditions, human interaction etc). and (d) physical obstacles blocking communication.

The protocol evolves in two phases:

> **Phase 1: The "Front" Creation Phase.** Initially the protocol builds (by using a limited, in terms of rounds, flooding) a sufficiently large "front" of particles, in order to guarantee the survivability of the data propagation process. During this phase, each particle having received the data to be propagated, deterministically forwards them towards the sink. In particular, and for a sufficiently large number of steps $s = 180\sqrt{2}$, each particle broadcasts the information to all its neighbors, towards the sink. Remark that to implement this phase, and in particular to count the number of steps, we use a counter in each message. This counter needs at most $\lceil \log 180\sqrt{2} \rceil$ bits.
>
> **Phase 2: The Probabilistic Forwarding Phase.** During this phase, each particle P possessing the information under propagation, calculates an angle ϕ by calling the subprotocol "ϕ-calculation" (see description below) and broadcasts $info(\mathcal{E})$ to all its neighbors with probability $\mathbb{P}_{fwd}$ (or it does not propagate any data with probability $1 - \mathbb{P}_{fwd}$) defined as follows:
>
> $$\mathbb{P}_{fwd} = \begin{cases} 1 & \text{if } \phi \geq \phi_{threshold} \\ \frac{\phi}{\pi} & \quad\text{otherwise} \end{cases}$$
>
> where ϕ is the angle defined by the line EP and the line PS and $\phi_{threshold} = 134^o$ (the selection reasons of this $\phi_{threshold}$ will become evident in Section 3.4).

In both phases, if a particle has already broadcast $info(\mathcal{E})$ and receives it again, it ignores it. Also the PFR protocol is presented for a single event tracing. Thus no multiple paths arise and packet sizes do not increase with time.

Remark that when $\phi = \pi$ then P lies on the line ES and vice-versa (and always transmits).

If the density of particles is appropriately large, then for a line ES there is (with high probability) a sequence of points "closely surrounding ES" whose angles ϕ are larger than $\phi_{threshold}$ and so that successive points are within transmission range. All such points broadcast and thus essentially they follow the line ES (see Fig. 4).

The ϕ-calculation subprotocol (see Fig. 5)
Let P_{prev} the particle that transmitted $info(E)$ to P.

(1) When P_{prev} broadcasts $info(E)$, it also attaches the info $|EP_{prev}|$ and the direction $\overrightarrow{P_{prev}E}$.

(2) P estimates the direction and length of line segment $P_{prev}P$, as described in the model.

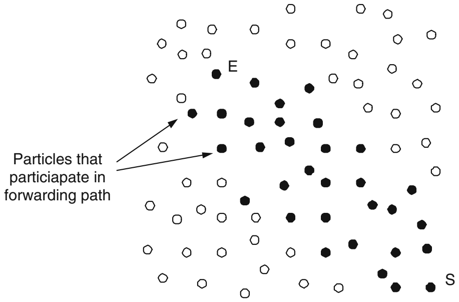


Fig. 4. Thin Zone of particles

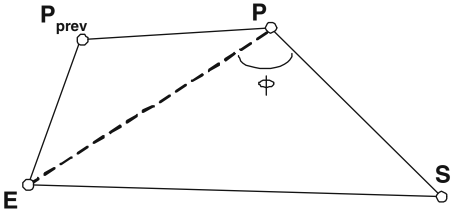


Fig. 5. Angle ϕ calculation example

(3) P now computes angle $(\widehat{EP_{prev}P})$, and computes $|EP|$ and the direction of $\overrightarrow{PE}$ (this will be used in further transmission from P).

(4) P also computes angle $(\widehat{P_{prev}PE})$ and by subtracting it from $(\widehat{P_{prev}PS})$ it finds ϕ.

Notice the following:

(i) The direction and distance from activated sensors to E is inductively propagated (i.e. P becomes P_{prev} in the next phase).

(ii) The protocol needs only messages of length bounded by $\log A$, where A is some measure of the size of the network area, since (because of (i) above) there is no cumulative effect on message lengths.

Essentially, the protocol captures the intuitive, deterministic idea "if my distance from ES is small, then send, else do not send". We have chosen to enhance this idea by random decisions (above a threshold) to allow some local flooding to happen with small probability and thus to cope with local sensor failures.

3.3 Properties of PFR

Any protocol Π solving the data propagation problem must satisfy the following three properties:

- **Correctness.** Π must guarantee that data arrives to the position S, given that the whole network exists and is operational.
- **Robustness.** Π must guarantee that data arrives at enough points in a small interval around S, in cases where part of the network has become inoperative.
- **Efficiency.** If Π activates k particles during its operation then Π should have a small ratio of the number of activated over the total number of particles $r = \frac{k}{N}$. Thus r is an energy efficiency measure of Π.

We show that this is indeed the case for PFR.

Consider a partition of the network area into small squares of a fictitious grid G (see Fig. 6). Let the length of the side of each square be l. Let the number of squares be q. The area covered is bounded by ql^2. Assuming that we randomly throw in the area at least $\alpha q \log q = N$ particles (where $\alpha > 0$ a suitable constant), then the probability that a particular square is avoided tends to 0. So with very high probability (tending to 1) all squares get particles.

We condition all the analysis on this event, call it F, of at least one particle in each square.

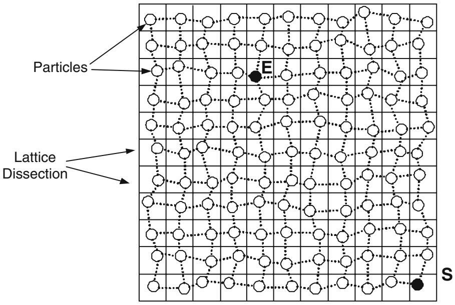


Fig. 6. A Lattice Dissection G

3.4 The Correctness of PFR

Without loss of generality, we assume each square of the fictitious lattice G to have side length 1.

We have proved the correctness of the PFR protocol, by using a geometric analysis. We below sketch the proof.

Consider any square Σ intersecting the ES line. By the occupancy argument above, there is with high probability a particle in this square. Clearly, the worst case is when the particle is located in one of the corners of Σ (since the two corners located most far away from the ES line have the smallest ϕ-angle among all positions in Σ).

By some geometric calculations, we finally prove that the angle ϕ of this particle is $\phi > 134^o$. But the initial square (i.e. that containing E) always broadcasts and any intermediate intersecting square will be notified (by induction) and thus broadcast because of the argument above. Thus the sink will be reached if the whole network is operational.

Lemma 4. PFR *succeeds with probability 1* in sending the information from E to S given the event F.

3.5 The Energy Efficiency of PFR

We consider the fictitious lattice G of the network area and let the event F hold. There is (at least) one particle inside each square. Now join all nearby particles of each particle to it, thus by forming a new graph G' which is "lattice-shaped" but its elementary "boxes" may not be orthogonal and may have varied length. When G's squares become smaller and smaller, then G' will look like G. Thus, for reasons of analytic tractability, we assume that particles form a lattice (see Fig. 7). They also assume length $l = 1$ in each square, for normalization purposes. Notice however that when $l \to 0$ then "$G' \to G$" and thus all results in this Section hold for any random deployment "in the limit".

The analysis of the energy efficiency considers particles that are active but are as far as possible from ES. Thus the approximation is suitable for remote particles.

We estimate an upper bound on the number of particles in an $n \times n$ (i.e. $N = n \times n$) lattice. If k is this number then $r = \frac{k}{n^2}$ $(0 < r \leq 1)$ is the "energy efficiency ratio" of PFR.

More specifically, we prove the (very satisfactory) result below. They consider the area around the ES line, whose particles participate in the propagation process. The number of active particles is thus, roughly speaking, captured by the size of this area, which in turn is equal to $|ES|$ times the maximum distance from $|ES|$ (where maximum is over all active particles).

This maximum distance is clearly a random variable. To calculate the expectation and variance of this variable, we basically "upper bound" the stochastic process of the distance from ES by a random walk on the line, and subsequently "upper bound" this random walk by a well-known stochastic process (i.e. the "discouraged arrivals" birth and death Markovian process, see e.g. [14]). Thus we can prove the following:

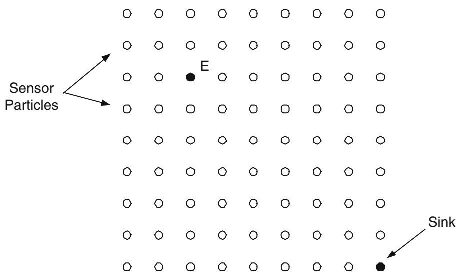


Fig. 7. A Lattice Sensor Network

Theorem 1. The energy efficiency of the PFR protocol is $\Theta\left(\left(\frac{n_0}{n}\right)^2\right)$ where $n_0 = |ES|$ and $n = \sqrt{N}$, where N is the number of particles in the network. For $n_0 = |ES| = o(n)$, this is $o(1)$.

3.6 The Robustness of PFR

To prove the following robustness result, we consider particles "very near" to the ES line. Clearly, such particles have large ϕ-angles (i.e. $\phi > 134^o$). Thus, even in the case that some of these particles are not operating, the probability that none of those operating transmits (during the probabilistic phase 2) is very small. Thus, we can prove the following:

Lemma 5. PFR manages to propagate the crucial data across lines parallel to ES, and of constant distance, with *fixed* nonzero probability (not depending on n, $|ES|$).

4 NanoPeer Networks and P2P Worlds

The peer-to-peer (P2P) paradigm has emerged to be one of the hottest subjects of research and development of computer science during the last few years. After the advent of Napster ([17]) and Seti@Home ([19]), researchers have focused on the fields of content and resource sharing P2P systems, usually dealing with such tasks as indexing and searching, routing, security and anonymity, resource exploitation and load balancing, etc. This is a natural consequence of the widespread use of such systems.

However, what researchers usually take for granted (i.e. average processing/storage/network capacities and power supply of modern computers) may not exist when we take a step further and deal with devices other than personal computers. Such restrictions may include little or no storage capacity or memory at the peers, highly unstable communication links, and power consumption issues, usually inherent in the fields of embedded devices, sensor networks, and ubiquitous computing in general.

We examine Smart Dust systems ([4]) - an inherently pure P2P system - and present NanoPeer Networks: an approach to P2P networks comprised of microdevices acting as lightweight peers in a P2P overlay, with restricted computing and energy capabilities. We try to identify the problems arising when applying computer-world techniques to this nano-world, attempt to locate the cause of such discrepancies, and propose outlines of relevant solutions.

We further argue that, due to the analogy between sensor networks and pure P2P systems, experience gathered through research and experimentation in the P2P field, can be indispensable when dealing with real-world problems in the nano-level. As we'll see, many of the issues arising when dealing with NanoPeers, have a computer-world counterpart which has already been dealt with by P2P scientists, thus making computer-world P2P systems a first-class testbench for nano-level solutions. This NanoPeers paradigm was first proposed in [21].

4.1 Critique: Intersection of P2P Worlds

What happens if we try to apply computer-world methods and protocols in the P2P world of SDCs? As already stated, existing routing/communication protocols are *not* appropriate for such devices.

First of all, available processing capabilities (i.e. CPU, memory, etc.) are very restricted when it comes to "grain" particles. DHT-based routing protocols require a minimum of O(log n) neighbors, while flooding translates to high overall power consumption. Remember, moreover, that Smart Dust systems are usually at the extreme of having practically *no* memory at all.

Second, particles are highly dynamic; they may "sleep" or fail at will. This, coupled with the restricted available memory, gives rise to new problems. For example, in a SDC, neighbor discovery must be done at every search operation, also taking into consideration power consumption issues.

Last but not least, the connectivity of particles is also restricted; a particle can contact only particles lying within a specific area. For one of these NanoPeers to access another peer outside its area of coverage, relaying-like solutions are required ([9]).

4.2 Smart-Dust and P2P Applications: Parallel Universes?

As pointed out in relevant literature ([2]), some of the main factors influencing the design and deployment of a sensor network, include fault-tolerance, scalability, hardware constraints, the network topology, the transmission media, and power consumption (as dictated by communication and data processing requirements). Sounds familiar? Let's take a step deeper and have a look at the dominant characteristics of SDCs and sensor networks (SNs) in general:

- The number of sensor nodes in a SN is very high – several orders of magnitude higher than the nodes in simple ad-hoc networks.
- Sensor nodes in a SN are deployed in a quite dense manner – as high as 20 sensor nodes per m^3. This, coupled with the fact that SN nodes mainly use broadcast communication, with a transmission range in the order of some meters, gives us a very dense, high-outdegree graph of sensor nodes and their interconnections.
- SN nodes are prone to failures of any kind, ranging from a simple power failure to the complete destruction of nodes by external factors (e.g. hostile action, dire environmental conditions, etc.).
- Sensor nodes have limited computing capabilities and power consumption capacity. For example, SDC particles may be at the extreme of having no memory at all.
- Bandwidth resources are also scarce. SDC particles are usually equipped with transmitters with transmission rates in the order of tens of kbps, although it's possible to use faster but greedier, with regards to energy consumption, transmitters (e.g. Bluetooth can achieve a 1Mbps transmission rate, for an energy consumption high enough to prevent it from being used in SDC).
- The topology of SNs may change very frequently, especially when sensor nodes are attached to mobile objects, or when they are deployed in an open

environment, with multiple moving obstacles in the way. Node failures also result in topology changes, albeit these are permanent ones.

Compare the above to the status-quo of the Gnutella peer-to-peer network overlay:

- The number of nodes in the Gnutella network is very high – several orders of magnitude higher than nodes in traditional distributed systems (e.g. Mosix [16], Beowulf [3], etc.).
- Nodes in Gnutella are interconnected according to a power-law topology, following the small-world paradigm. Due to the quasi-complete connectivity of the underlying TCP/IP network, Gnutella nodes may have multiple neighbor nodes. Moreover, Gnutella also uses broadcasting (flooding) techniques to propagate information through the overlay.
- Gnutella nodes are selfish ([1]); nearly 70% of the nodes share no files with the rest of the community. Without loss of generality, we can models such peers as computing entities of ultra-low computing capabilities, much like nodes in a sensor network.
- Following the above distribution, most Gnutella nodes have very low connection speeds (in the order of tens of kbps), while there do exist some (but few) nodes with high-bandwidth connections (e.g. 1Mbps lines).
- "Free-riders", are usually users entering the Gnutella overlay over dial-up modem lines, and may exit the overlay without prior notice (e.g. due to a modem hang-up). This makes such nodes highly volatile and results in a frequently changing network topology. For example, [22] has shown that the half-life – the time required for half of the peer population to be replaced due to joins and leaves – of MojoNation ([15]) was less than one hour!

The similarities of sensor networks and pure P2P computer-world systems extend well beyond the above mentioned characteristics. For example, Gnutella entered the realm of hybrid P2P systems, with the introduction of "UltraPeers" by LimeWire; in about the same time, researchers in the sensor networks' field proposed a "backbone"-based architecture ([8]), where some "higher-order" sensor nodes were injected into a sensor network and took over the communication tasks.

However, as already mentioned, there exist more than a handful of differences between the two worlds. For one, computers have no energy limitations (other than that they rely on the public electricity network's being stable) and their storage capacity and computing capabilities are growing at a quasi-exponential rate. SDC particles, bounded by the limitations in size and energy, don't (and probably will never) have access to such equipment as a multi-gigabyte hard disk or a multi-gigahertz CPU or a multi-megabyte RAM.

5 Smart-Dust NanoPeers: One Step Further

So far, the SDC model features a single "wall" or "sink". We hereby propose some extensions to this model, borrowed from the computer-world P2P experience.

5.1 Multi-“wall”/“Sink” Smart Dust Systems

Computer-world P2P systems use a set of servers, scattered around the net, to handle authentication and bootstrapping of new nodes. Thus, we can imagine a SDC with multiple walls (e.g. particles “hovering” in a cube, whose all six sides are walls). Note that the SDC model assumes that all particles find out about the position of their wall during bootstrapping.

Multiple-sink systems have been also studied in [11], although their system is pull-based; the walls send out queries towards an area of interest – thus creating (possibly several) path(s) of hops on the smart dust plane – and replies are piggybacked on this very path. What happens, though, if we are interested in a push-based system (i.e. in a system were it’s the particles that decide when they have something important to say to the rest of the world)?

A first naive solution to the multiple-wall problem is for the wall(s) to inform particles of their existence, in a P2P recursive manner: every wall registers with all particles within reach, and registration information is propagated recursively, using the inexpensive digital radio transceivers. Particles are then responsible for selecting the wall that is closer to them.

5.2 P2P Worlds: A Hybrid Model

A better and more scalable solution to the problems that may arise in the SDC P2P world, would be to have a Hybrid P2P system, consisting of heterogeneous particles, with escalating processing capabilities, network bandwidth, area of coverage, and power supply, allowing for multiple levels of peers.

In this scenario, every set of homogeneous particles would form a separate *Smart Dust Layer* (or *SDL*). Higher order SDLs would then act as “walls” for lower order SDLs, with the actual wall(s) being seamlessly incorporated in this model. Imagine such a world were micro-peers coordinate the operation of nano-peers within their area of coverage, milli-peers coordinate micro-peers etc.

Particles would then contact the higher-order particle that is closer to them (discovered via the *broadcast beacon* mode). To go one step further, we can have particles use only the low-consumption digital radio transceiver to broadcast observations, under the virtual “umbrella” of one or more higher-order particles, much in the way overlapping GSM cells operate.

5.3 An Example Application of P2P Worlds

Parcicles: Trails on Smart-Dust Suppose that $Very\ Important\ Parcels^{TM}$ (VIP^{TM}), a (quite imaginary) major postal delivery firm, implants in every parcel shipped a *parcicle* - a low-cost, cut-down version of a conventional particle (i.e. a NanoPeer), with all sensors stripped-off, featuring only the communication equipment, plus a unique id number (given to customers along with their receipts).

Imagine that each box carrying parcels features a “higher-order” particle (i.e. an MicroPeer), or a set of such particles, equipped with a better battery and some memory. Let’s assume that more such particles (i.e. MilliPeers) are spread around VIP^{TM}'s warehouses, with each warehouse having a set of servers (e.g. one per department), and with all of these servers across all warehouses being interconnected in a P2P network. Assume that *parcicles* periodically register with their box's particle(s), with registration information being propagated and cached all the way up to the local servers.

Suppose now that a customer of VIP^{TM}, wishes to track down a parcel's current position. She would then utilize the all-secure $VIP - Tracker^{TM}$ P2P software to query the servers' P2P overlay for the one having last seen her parcel. Verification of the parcel's actual position could be done on-the-fly, with this last server sending an "are-you-there" query all the way down to the parcel's *parcicle* (or to the particle caching the *parcicle*'s registration, should the latter be asleep). Note that there is both horizontal and vertical communication, within and across the layers of the architecture, in order to register information and to answer queries.

References

1. E. Adar and B. Huberman: Free riding on Gnutella. Technical report, Xerox PARC, 2000.
2. I.F. Akyildiz, W. Su, Y. Sankarasubramaniam and E. Cayirci: Wireless sensor networks: a survey. In the Journal of Computer Networks, Volume 38, pp. 393-422, 2002.
3. Beowulf. http://www.beowulf.org/.
4. Berkeley Wireless Research Center. http://bwrc.eecs.berkeley.edu/.
5. A. Boukerche and S. Nikoletseas: Protocols for Data Propagation in Wireless Sensor Networks: A Survey. Chapter in the Book "Wireless Communications Systems and Networks", Editor Mohsen Guizani, Kluwer Academic Publishers, Date Published: 06/2004, ISBN: 0306481901, 718 p.
6. A. Boukerche and S. Nikoletseas: Energy Efficient Algorithms in Wireless Sensor Networks. Invited Book Chapter, Springer Verlag, to appear in 2004.
7. I. Chatzigiannakis, T. Dimitriou, S. Nikoletseas and P. Spirakis: A Probabilistic Algorithm for Efficient and Robust Data Propagation in Smart Dust Networks. In the Proceedings of the 5th European Wireless Conference on Mobile and Wireless Systems beyond 3G (EW 2004), pp. 344-350, 2004. Also, invited paper in the Journal of Adhoc Networks.
8. I. Chatzigiannakis, S.Nikoletseas, and P.Spirakis: Distributed communication algorithms for ad-hoc mobile networks. In the Journal of Parallel and Distributed Computing (JPDC), Special Issue on Mobile Ad-hoc Networking and Computing, 63 (2003) 58-74, 2003.
9. I. Chatzigiannakis, S. Nikoletseas, and P. Spirakis: Smart Dust Protocols for Local Detection and Propagation. In Proc. of Principles of Mobile Computing (POMC), ACM Press, pp. 9-16, 2002. Also, in the ACM/Baltzer MONET Journal, Special issue on Algorithmic Solutions for Wireless, Mobile, Ad Hoc and Sensor Networks, accepted, to appear in 2004.
10. D. Estrin, R. Govindan, J. Heidemann and S. Kumar: Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proc. 5th ACM/IEEE International Conference on Mobile Computing* – MOBICOM'1999.

11. C. Intanagonwiwat, R. Govindan and D. Estrin: Directed Diffusion: A Scalable and Robust Communication Paradigm for Sensor Networks. In *Proc. 6th ACM/IEEE International Conference on Mobile Computing* – MOBICOM'2000.
12. C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann and F. Silva: Directed Diffusion for Wireless Sensor Networking. Extended version of [11].
13. J.M. Kahn, R.H. Katz and K.S.J. Pister: Next Century Challenges: Mobile Networking for Smart Dust. In *Proc. 5th ACM/IEEE International Conference on Mobile Computing*, pp. 271-278, September 1999.
14. L. Kleinrock: Queueing Systems, Theory, Vol. I, pp. 100. *John Wiley & Sons*, 1975.
15. Autonomous Zone Industries. Mojonation. http://www.mojonation.com/.
16. Mosix. http://www.mosix.org/.
17. Napster. http://www.napster.com/.
18. S. M. Ross: Stochastic Processes, 2nd Edition. *John Wiley and Sons, Inc.*, 1995.
19. Seti@Home. http://setiathome.ssl.berkeley.edu/.
20. P. Spirakis: Algorithmic and Foundational Aspects of Sensor Systems. Invited talk at the 1st International Workshop on Algorithmic Aspects of Wireless Sensor Networks (ALGOSENSORS 04), Eds S. Nikoletseas and J. Rolim LNCS 3121, Springer Verlag, 2004.
21. P. Triantafillou, N. Ntarmos, S. Nikoletseas, and P. Spirakis: NanoPeer Networks and P2P Worlds. In Proc. 3rd IEEE International Conference on Peer-to-Peer Computing (P2P 2003), September 2003.
22. B. Wilcox-O'Hearn: Experiences deploying a large-scale emergent network. In *Proc. of IPTPS '02.*

The Kell Calculus: A Family of Higher-Order Distributed Process Calculi

Alan Schmitt and Jean-Bernard Stefani

INRIA Rhône-Alpes 655 Avenue de l'Europe,
Montbonnot, 38334 St Ismier, France

Abstract. This paper presents the Kell calculus, a family of distributed process calculi, parameterized by languages for input patterns, that is intended as a basis for studying component-based distributed programming. The Kell calculus is built around a π-calculus core, and follows five design principles which are essential for a foundational model of distributed and mobile programming: hierarchical localities, local actions, higher-order communication, programmable membranes, and dynamic binding. The paper discusses these principles, and defines the syntax and operational semantics common to all calculi in the Kell calculus family. The paper provides a co-inductive characterization of contextual equivalence for Kell calculi, under sufficient conditions on pattern languages, by means of a form of higher-order bisimulation called strong context bisimulation. The paper also contains several examples that illustrate the expressive power of Kell calculi.

Keywords: Process calculi, distributed programming, mobile code, ambients, components, higher-order languages, higher-order bisimulation.

1 Introduction

Wide-area distributed systems and their applications are increasingly built as heterogeneous, dynamic assemblies of software components. Such components can be downloaded from different sources, can spontaneously interact across the Internet, and may fail, maliciously or accidentally, in many ways. In that context, a model for wide-area distributed programming should provide the means to describe and reason about such component assemblies, as well as to control their dynamic configuration (which components to connect, which components to sandbox for security, from which trusted location to download a component, which faulty component to replace, etc).

In the past fifteen years or so, there have been numerous works aiming at defining wide-area distributed programming models, especially in the area of distributed process calculi (see e.g. [11, 14] for recent surveys of distributed process calculi and process calculi with localities), as well as in the area of component-based programming and software architecture (see e.g. [21] for a recent overview of formal approaches to component-based programming). Distributed process calculi are especially interesting as foundations for distributed programming for they allow to formally elicit, in a concise form, key operators and programming primitives which can then be analyzed in different ways (expressive power, semantical equivalences, etc), and which serve as a basis for additional developments (type systems, specification logics, and verification tools).

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 146–178, 2004.

In the production of the last fifteen years, we can single out three major kinds of distributed process calculi:

1. Variants of the first-order π-calculus with localities and migration primitives, including notably the Distributed Join calculus [16, 15], the Dπ calculus [19], lsdπ [30], Nomadic Pict [38], and the Seal calculus [10], .
2. Variants of Mobile Ambients, including notably the original Mobile Ambients [9], Safe Ambients [23], Safe Ambients with passwords [26], Boxed Ambients [6], New Boxed Ambients [7], the M^3 calculus [13], and the calculus of Bounded Capacities [3].
3. Distributed higher-order calculi, including Facile/CHOCS [22, 36], higher-order variants of Dπ [39, 18], Klaim [28, 29], and the M-calculus [33].

Interestingly, the first two kinds of calculi above share the same concern: allowing process mobility while avoiding an explicitly higher-order calculus, on the grounds that the semantical theory of higher-order process calculi is much harder than that of first-order calculi. We shall argue below that such a concern is ill-founded, for two main reasons. First, interesting reconfiguration phenomena in distributed assemblies of components are inherently higher-order and can be given a more direct account using higher-order communications. Second, process mobility remains an inherently higher-order operation with the same semantical difficulties.

Among higher-order calculi, the M-calculus is the only one, to our knowledge, to provide hierarchical and programmable localities. We argue below that such localities are necessary to account for notions of components, as well as to provide extensive isolation capabilities which are essential for security and fault handling. Unfortunately, with its multiple routing rules, the M-calculus is rather complex, and developing directly a suitable bisimulation theory for it seems a daunting task. The Kell calculus we study in this paper has been developed both as a simplification and a generalization of the M-calculus. In particular, the Kell calculus has simple communication rules across locality boundaries, which subsume the complex routing rules of the M-calculus. Also, the Kell calculus is in fact a family of higher-order process calculi with localities, which share the same basic operational semantics rules, but differ in the language used to define receipt patterns in input constructs.

The simpler constructs of the Kell calculus (compared to those of the M-calculus) allow us to develop a theory of strong bisimulation and a natural notion of contextual equivalence for the Kell calculus. Under sufficient conditions on pattern languages, we obtain a co-inductive characterization of contextual equivalence in terms of a form of higher-order bisimulation, which, following Sangiorgi, we call strong context bisimulation. Interestingly, strong context bisimulation for the Kell calculus is no more complex than higher-order bisimulations that have been proposed for Ambient calculi and for the Seal calculus. This lends support to our claim that the semantical theory of higher-order process calculi need not be more complicated than for first-order process calculi with mobility.

This paper is organized as follows. In Section 2, we review the main considerations behind the different constructs of the Kell calculus, and discuss related work in that light. In section 3 we define the syntax and operational semantics of the Kell calculus, and prove that the reduction semantics coincides with a labelled transition system semantics, under sufficient conditions on the pattern language used. In section 4, we introduce a strong contextual equivalence for the Kell calculus which we prove to coincide with strong context bisimulation. In section 5, we discuss several instances of Kell calculi, and give several examples that illustrate the expressive power of these calculi. Section 6 concludes the paper with a discussion of further work.

2 Design Principles

The Kell calculus is built around a π-calculus core, and obeys five main design principles which we consider important for a foundational model of distributed programming: higher-order communications, hierarchical localities, programmable membranes, local actions, and dynamic binding. In this section we motivate these design principles, we introduce informally the main constructs of the Kell calculus, and we discuss related work in the light of these principles.

π-Calculus Core. The π-calculus is a standard of expressivity and a reference for concurrent computing. The Kell calculus has the same basic constructs than the π-calculus. In particular, we have:

- Binary communication on named channels: an output on channel a is noted $a\langle w\rangle.T$, where w is an argument, and T is continuation; an input on channel a can take the form $\xi \triangleright P$, where ξ is an input pattern (e.g. $a\langle x\rangle$, with x an input variable), and P is a process.
- Parallel composition of processes: $(P \mid Q)$ denotes the parallel composition of processes P and Q.
- Restriction: $\nu a.P$ denotes the restriction of name a in process P (or, equivalently, the creation of a fresh name a whose initial scope is P).
- A null process, $\mathbf{0}$, that does not perform any action.

The Kell calculus family arises from different choices for the language of input patterns. Keeping the Kell calculus parametric in the language of input patterns, yielding a family of Kell calculi, is motivated by two main reasons:

1. Calculi of different expressive power can be obtained by varying the language of input patterns. For instance, [8] introduces a π-calculus with structured channel names and proves that the π-calculus variant obtained is strictly more expressive than the (synchronous) π-calculus. We present in the paper several Kell calculus instances, with increasing expressive power. By just varying the pattern language, we obtain calculi with name-passing, name matching and name-unmatching operators, polyadic communications, Join calculus-like patterns, and tuple-matching.
2. Sophisticated pattern matching capabilities are extremely useful in practical programming languages, as demonstrated by pattern matching constructs in functional languages such as OCaml. Jocaml and Polyphonic C# highlight the usefulness of Join-calculus-like input patterns for practical concurrent programming. By allowing pattern matching in Kell calculus input constructs, we can provide foundations for distributed programming languages with filtering. Abstracting the pattern language allows us to derive results (e.g. for the characterization of contextual equivalence) that hold for a variety of Kell calculi, with different filtering characteristics.

Hierarchical Localities. Hierarchical localities are characteristic of the Distributed Join calculus and of Ambient calculi. Localities in the Kell calculus are noted, classically, $a[P]$, where a is the name of the locality, and P is a process. Intuitively, $a[P]$ can be read "process P executes at location a", or "$a[P]$ is a component named a, with state (or behavior) P". A process P can in turn comprise a parallel composition of named localities. A named locality $a[P]$ is the Kell calculus is called a *kell*[1].

The introduction of hierarchical localities in the Kell calculus is motivated by two main considerations:

1. The hardware and software structure of distributed systems almost always takes the form of hierarchies of (named) localities. For instance, a wide area network connects local-area networks, which comprise several machines, which execute several virtual machines (residing in different user address spaces), that provide different thread pools for running separate applications. This hierarchy constitutes a containment hierarchy with different failure semantics and control semantics. For instance, a failure of the wide-area network may prevent communication between local-area networks but does not hamper communication internal to a local-area network. In contrast, the failure of a machine will cause the failure of all the virtual machines that it executes. Also, a machine (or its operating system) may kill any of the virtual machines it runs, whereas a local area network cannot shut down any of the machines it hosts. Named localities in the Kell calculus provide a way to model such a hierarchy. This is useful to develop distributed management applications, concerned e.g. with component deployment, security management (including building firewalls for access control), and fault management (including isolating faulty subsystems). The approach to the secure execution of un-trusted components by means of wrappers championed by Boxed-π [34] illustrates the use of named hierarchical localities for isolation purposes.
2. The structure of software systems built by assembly of software components can be modelled as hierarchies of named localities. We illustrate this in Section 5, with the sketch of the modelling in the Kell calculus of a hierarchical component model, representative of recent work on architecture description languages (see [25] for a recent survey of architecture description languages). Allowing both sharing and containment in software structures has for a long time been recognized as an important requirement in object-oriented programming, as witnessed by the numerous works dealing with containment types for object systems (see e.g. [12] for a recent work on this subject, and [1] for a use of containment types in a Java-based software component model).

Interestingly, apart from the M-calculus, no higher-order distributed process calculus provides hierarchical localities.

Local Actions. In the Kell calculus, computing actions can take four simple forms, illustrated below with simple patterns:

[1] The word "kell" is intended to remind of the word "cell", in a loose analogy with biological cells.

1. Receipt of a local message, as in the reduction below, where a message, $a\langle Q\rangle.T$, on port a, bearing the process Q and the continuation T, is received by the trigger $a\langle x\rangle \triangleright P$:
$$a\langle Q\rangle.T \mid (a\langle x\rangle \triangleright P) \;\rightarrow\; T \mid P\{Q/x\}$$
2. Receipt of a message originated from the environment of a kell, as in the reduction below, where a message, $a\langle Q\rangle.T$, on port a, bearing the process Q and contination T, is received by the trigger $a\langle x\rangle^{\uparrow} \triangleright P$, located in kell b :
$$a\langle Q\rangle.T \mid b\left[a\langle x\rangle^{\uparrow} \triangleright P\right].S \;\rightarrow\; T \mid b\left[P\{Q/x\}\right].S$$
In input pattern $a\langle x\rangle^{\uparrow}$, the arrow $^{\uparrow}$ denotes a message that should come from the outside of the immediately enclosing kell.
3. Receipt of a message originated from a sub-kell, as in the reduction below, where a message, $a\langle Q\rangle.T$, on port a, bearing the process Q and continuation T, and coming from sub-kell b, is received by the trigger $a\langle x\rangle^{\downarrow} \triangleright P$, located in the parent kell of kell b:
$$(a\langle x\rangle^{\downarrow} \triangleright P) \mid b\left[a\langle Q\rangle.T \mid R\right].S \;\rightarrow\; P\{Q/x\} \mid b\left[T \mid R\right].S$$
In pattern $a\langle x\rangle^{\downarrow}$, the arrow $^{\downarrow}$ denotes a message that should come from the inside of a sub-kell.
4. Passivation of a kell, as in the reduction below, where the sub-kell named a is destroyed, and the process Q it contains is used in the guarded process P:
$$a\left[Q\right].T \mid (a\left[x\right] \triangleright P) \;\rightarrow\; T \mid P\{Q/x\}$$

Actions of the form 1 above are standard π-calculus actions. Actions of the form 2 and 3 are just extensions of the message receipt action of the π-calculus to the case of triggers located inside a kell. They can be compared to the communication actions in the Seal calculus and in the Boxed Ambients calculus. Actions of the form 4 are peculiar to the Kell calculus. They allow the environment of a given kell to exercise control over the execution of the process located inside this kell. Notice that the construct $a[P].Q$ plays a dual role: that of a locus of computation since process P may execute within a and receive messages from the environment of a, and that of a message that may be consumed.

The different actions in the Kell calculus are all *local* actions. This means that atomic actions in the calculus occur entirely within the context of a locality, or at the boundary between a locality and enclosing environment. Let us explain in detail what this means. Consider the configuration $C \triangleq c[a[P] \mid b[Q] \mid S \mid \ldots]$. Localities $a[P]$ and $b[Q]$, which act as separate loci of computation, share a global communication medium, represented by the enclosing locality named c. The locality c can be considered a model of some communication network or computational environment, for its sub-localities.

What the principle of local actions forbids, if localities are supposed to model distinct locations in network space, is an event that would involve two remote localities a and b in one atomic action, *without the mediation of the enclosing locality*. For instance, the locality principle would forbid a reduction such as the following one, which is reminiscent of the `in` primitive in Mobile Ambients):

$$(\dagger) \qquad a\left[c\langle P\rangle \mid R\right] \mid b\left[(c\langle x\rangle \triangleright x) \mid Q)\right] \rightarrow a\left[R\right] \mid b\left[P \mid Q\right]$$

Interestingly, note that, by the same reasoning, we can consider as *local* a reduction of the form:

$$(\dagger\dagger) \qquad a[c\langle P\rangle \mid R] \mid b[Q] \mid (c\langle x\rangle^{\downarrow} \mid b[y] \triangleright b[x \mid y]) \rightarrow a[R] \mid b[P \mid Q]$$

Even though the effect is similar to that of the reduction $(\dagger)$, it is important to note here that the transfer of process P from a to b is in fact mediated by the common environment shared by a and b. The atomic transfer between two adjacent domains is thus a property of their environment (which may or may not model a realistic network). In fact what is problematic with the reduction $(\dagger)$ is the fact that it *mandates* the existence of a network environment where atomic transitions between potentially spatially remote nodes are possible, thus excluding e.g. wide area networks. In contrast, reduction $(\dagger\dagger)$ merely specifies a particular network environment which happens *not* to be a wide area network. With this approach, one can model different forms of computational environments, which may or may not correspond to wide area networks, but *one is neither forced to consider atomic actions that need to occur across wide-area networks, nor forced to only use purely asynchronous communications in all localities*.

The principle of local actions is motivated by the inherent limitations and costs involved in achieving atomic communication in an asynchronous network environment. Strictly speaking, an atomic communication primitive (communication takes place in full, and senders and recipients are aware of it, or not all) would entail common knowledge (consensus), between senders and recipients, of the outcome of a successful communication, a situation which is known impossible to achieve in a purely asynchronous network with failures [24]. Probabilistic protocols can be used to get arbitrarily close (in probability) to the ideal situation, but they are much more costly to implement than the simple sending of a message, and they rely on assumptions about the knowledge of each site on fault occurrences in a given configuration which are not necessarily valid in a wide-area network. In a *foundational* calculus for distribution, adopting atomic remote communication primitives would mean that a simple information exchange would have to bear the cost of the implied atomicity. This is clearly not acceptable, for there are useful programs that can be built relying on communication primitives with weak guarantees. For instance, an application that monitors periodically a large number of sensors distributed throughout a wide-area network can make use directly of a protocol with weak guarantees such as the UDP Internet protocol. Relying on atomic communication primitives for these applications would result in higher performance costs for no added benefit. Besides, since in a wide-area network with purely asynchronous behavior, failure cannot be distinguished from arbitrary long delays in communication, a practical program would encapsulate an atomic remote communication primitive in a timer watchdog, raising an exception if the communication takes too long to complete. In effect, this encapsulation exposes the more complex transactional behavior of communication, which comprises, from the point of view of the sender, two local actions: an initial local action (initiating the communication), and a terminal local action (aborting the communication, in case of communication failure or excessive delay, or reporting success).

Interestingly, the principle of local actions is not valid in Ambient calculi. Even the Boxed Ambient variants, which have been inspired by the Seal calculus (which *does* enforce the principle of local actions), still contain at least one atomic communication primitive: the `in` primitive. Of course, in Boxed Ambient calculi, one has also the first-order communication primitives of the Seal calculus at his disposal. Still, the atomic semantics of the `in` primitive is not satisfactory. As mentioned above, a more useful semantics for this primitive would turn it into a transaction, with at least two possible outcomes (success or failure), thus allowing the initiator of the communication to take into account the potential occurrence of failures.

Higher-Order Communication. Higher-order communication is an important feature of a distributed programming model for it allows to model distributed software updates, the introduction of new components in a system, and in general dynamic reconfiguration. As can be seen in the discussion of the different actions of the Kell calculus above, the Kell calculus supports higher-order communication.

In the first two categories of distributed process calculi we mentioned in the introduction (variants of the π-calculus with mobility primitives, Ambient calculi), dynamic reconfiguration is possible through mobility primitives, however we find that the modelling of dynamic reconfiguration behavior in these models can become a bit contrived.

Consider for instance the following creation of a new configuration in the Kell calculus:

$$a\langle P, Q, R\rangle \mid (a\langle x, y, z\rangle \triangleright b[x \mid c[y \mid e[z]]]) \;\rightarrow\; b[P \mid c[Q \mid e[R]]]$$

One can have an analogous creation of a new configuration in an Ambient calculus via a construction of the form

$$\langle\epsilon\rangle \mid M_b \mid M_c \mid M_e \mid (x).b[\mathtt{open}\, b \mid c[\mathtt{open}\, c \mid e[\mathtt{open}\, e]]] \;\rightarrow^*\; b[P \mid c[Q \mid e[R]]]$$

where $M_b \triangleq b[\mathtt{in}\, b.P]$, $M_c \triangleq c[\mathtt{in}\, b.\mathtt{in}\, c.Q]$, $M_e \triangleq e[\mathtt{in}\, b.\mathtt{in}\, c.\mathtt{in}\, e.Q]$. Notice, however, that this is not entirely sufficient to capture the atomic creation given in the Kell calculus above, for one needs to make sure that no parasitic move involves the b, c and e ambients once they are released after the initial communication. This leads to quite an involved protocol to perform the simple creation above. Another problem with such distributed calculi, except for the Seal calculus, is the absence of a replication facility. Thus, the following atomic action in the Kell calculus

$$a\langle P\rangle \mid (a\langle x\rangle \triangleright b[x] \mid c[x \mid d[x]]) \;\rightarrow\; b[P] \mid c[P \mid d[P]]$$

can only be obtained in an Ambient calculus with a protocol involving an a priori replication of P somewhere in the forest of ambients, and then routing the obtained copies to their proper destinations, e.g.

$$!(x).x.P \mid \langle\mathtt{in}\, b\rangle \mid \langle\mathtt{in}\, c\rangle \mid \langle\mathtt{in}\, c.\mathtt{in}\, d\rangle \mid (b[] \mid c[d[]]) \;\rightarrow^*\; b[P] \mid c[P \mid d[P]]$$

Again, ensuring the proper atomicity of the creation would involve additional subtlety for what is a fairly simple atomic creation in the Kell calculus.

In the Seal calculus, the presence of the migrate and replicate primitive simplifies the matter a bit, but the result is still fairly contrived compared to the simple reduction in the Kell calculus:

$$\overline{a}_1^*(e_1).\overline{a}_2^*(e_2).\overline{a}_3^c(e_3) \mid e_1[P] \mid e_2[P \mid a_3^{\uparrow}(d)] \mid a_1^*(b).a_2^*(c).e_3[P] \;\rightarrow^*\; b[P] \mid c[P \mid d[P]]$$

As with Ambients, the above Seal configuration would need to be more sophisticated to account for the atomic character of the original Kell calculus reduction.

The added complexity, in non-higher-order calculi, in expressing simple operations such as those above could be justified if this resulted in a simpler semantical theory. However, this is not the case: bisimulation equivalences defined for Ambient calculi and for the Seal calculus so far are definitely higher-order. For the Seal calculus we do not even have at this stage a sound and complete characterization of contextual equivalence. The higher-order bisimulation we develop in Section 4.2 for the Kell calculus is in fact no more complex than analogous bisimulations for Ambient calculi and the Seal calculus. In addition, it yields a sound and complete characterization of contextual equivalence (for a certain class of pattern languages). In our view, this removes a large part of the prevention against higher-order languages on the ground of the complexity of their bisimulation theory.

Finally, even though the Kell calculus has no primitive for recursion or replication, it is possible to define *receptive triggers*, i.e. triggers that are preserved during a reduction (much like definitions in the Join calculus) using its higher-order character. Let t be a name that does not occur free in ξ or P. We define $\xi \diamond P$ as follows:

$$\xi \diamond P \triangleq \nu t. Y(P, \xi, t) \mid t\langle Y(P, \xi, t)\rangle$$
$$Y(P, \xi, t) \triangleq \xi \mid t\langle y\rangle \, \triangleright \, P \mid y \mid t\langle y\rangle$$

It is easy to see with the rules of reduction given in Section 3 that if $M \mid (\xi \triangleright P) \rightarrow M' \mid P\theta$, where θ is a substitution, then we have $M \mid (\xi \diamond P) \rightarrow M' \mid (\xi \diamond P) \mid P\theta$. The construction $\xi \diamond P$ is reminiscent of the CHOCS fixed point operator defined in [36] and of Vasconcelos' fixed point operator in higher-order π [37].

Programmable Membranes. This principle refers to the ability to design kells with varying semantics, in order to reflect the different kinds of containment structures ("domains") that may arise in a distributed system (e.g. failure, security, or communication domains). Especially important for security is the ability to design different forms of control to gain access to the internals of a kell (firewalls), or, conversely, to restrict access from the internals of a kell to services provided by its environment (sandboxes and wrappers). The principle thus implies the ability to define arbitrary interface protocols to govern the communication between a locality and its environment, and the ability to control communications to and from the contents of a locality, much as is provided by the Seal calculus and calculi inspired by it such as Boxed Ambients and Boxed-π. The examples below show that the Kell calculus supports programmable membranes in the above sense. In contrast to Boxed-π and Boxed Ambients, the Kell calculus provides the ability to control the internals of a given kell by means of passivation actions. Such actions generalize the "migrate and replicate" constructs of the Seal calculus, while providing a way to recover the expressive power of the passivate operator of the M-calculus.

With the different forms of actions in the Kell calculus, one can program different forms of membranes. Here are some examples:

Perfect Firewall: Let P be an arbitrary process. The process $\nu e.e[b[P]]$ can only have internal transitions. The context $\nu e.e[b[\cdot]]$ is a *perfect firewall*. We shall prove in Section 4 that this is indeed so.

Transparent Membrane: Assume that all messages to process P located in a take the form $\mathtt{rcv}\langle a, Q\rangle$, that all messages sent by P to its environment take the form $\mathtt{snd}\langle c, Q\rangle$, and that there is an environment process $\mathtt{Env} \triangleq \mathtt{snd}\langle (a), x\rangle^{\downarrow} \diamond \mathtt{rcv}\langle a, x\rangle$ (this environment allows two processes located in different places to communicate, as in $a[P] \mid b[Q] \mid \mathtt{Env}_t$). Then, the process $\nu e.e[M_t \mid a[P]]$, where

$$M_t \triangleq (\mathtt{rcv}\langle a, x\rangle^{\uparrow} \diamond \mathtt{rcv}\langle a, x\rangle) \mid (\mathtt{snd}\langle (a), x\rangle^{\downarrow} \diamond \mathtt{snd}\langle a, x\rangle)$$

behaves exactly like $a[P]$ from the point of view of communications. Note that in the process definition above we use a receipt pattern $\mathtt{snd}\langle (a), x\rangle^{\downarrow}$ that makes use of a name variable (a), and of a process variable x, as well as a receipt pattern $\mathtt{rcv}\langle a, x\rangle^{\uparrow}$ that makes use of a name matching pattern a and of a process variable x. Pattern languages that support these constructs are defined in Section 5.

Intercepting Membrane: Under the same assumption as for the transparent membrane, the process $\nu e.e[M_i \mid a[P]]$, where

$$M_i \triangleq (\mathtt{rcv}\langle a, x\rangle^{\uparrow} \diamond FR(a, x)) \mid (\mathtt{snd}\langle (a), x\rangle^{\downarrow} \diamond FS(a, x))$$

defines a membrane around kell $a[P]$ that triggers behavior $FR(a, x)$ upon receipt of a message $\mathtt{rcv}\langle a, Q\rangle$, and behavior $FS(c, x)$ when P sends a message $\mathtt{snd}\langle c, Q\rangle$.

Migration Membrane: Under the same assumption as for the transparent membrane, the process $\nu e.e[M_m \mid a[P]]$, where

$$\begin{aligned} M_m \triangleq M_t &\mid (\mathtt{rcv}\langle a, \mathtt{enter}\langle x\rangle\rangle^{\uparrow} \mid a[y] \diamond a[y \mid x]) \\ &\mid (\mathtt{go}\langle (b)\rangle^{\downarrow} \mid a[y] \diamond \mathtt{snd}\langle b, \mathtt{enter}\langle a[y]\rangle\rangle) \end{aligned}$$

defines a membrane around kell $a[P]$ that allows it to move to a different kell via the go operation. Compare these operations with the migration primitives of Mobile Ambients, and the go primitive of Dπ or of the Distributed Join calculus.

Fail-Stop Membrane: Under the same assumption as for the transparent membrane, the process $\nu e.e[M_f \mid a[P]]$, where

$$\begin{aligned} M_f \triangleq M_t &\mid (\mathtt{rcv}\langle a, \mathtt{stop}\rangle^{\uparrow} \mid a[y] \triangleright S) \\ &\mid (\mathtt{rcv}\langle a, \mathtt{ping}\langle (r)\rangle\rangle^{\uparrow} \mid a[y] \diamond \mathtt{snd}\langle r, \mathtt{up}\rangle \mid a[y]) \\ S \triangleq {} & \mathtt{rcv}\langle a, \mathtt{ping}\langle (r)\rangle\rangle^{\uparrow} \diamond \mathtt{snd}\langle r, \mathtt{down}\rangle \end{aligned}$$

defines a membrane around kell $a[P]$ that allows to $\mathtt{stop}$ its execution (simulating a failure in a fail-stop model), and that implements a simple failure detector via the $\mathtt{ping}$

operation. Compare these operations with the π_{1l}-calculus [2] or the Distributed Join calculus models of failures.

Fail-Stop Membrane with Recovery: Under the same assumption as for the transparent membrane, the process $\nu e.e[M_r \mid a[P]]$, where

$$\begin{aligned} M_r &\triangleq M_t \mid (\mathtt{rcv}\langle a, \mathtt{stop}\rangle^{\uparrow} \mid a[y] \diamond b\langle y\rangle \mid T) \\ &\qquad \mid (\mathtt{rcv}\langle a, \mathtt{ping}\langle (r)\rangle\rangle^{\uparrow} \mid a[y] \diamond a[y] \mid \mathtt{snd}\langle r, \mathtt{up}\rangle) \\ T &\triangleq (\mathtt{rcv}\langle a, \mathtt{ping}\langle (r)\rangle\rangle^{\uparrow} \mid b\langle y\rangle \diamond b\langle y\rangle \mid \mathtt{snd}\langle r, \mathtt{down}\rangle) \\ &\qquad \mid (\mathtt{rcv}\langle a, \mathtt{recover}\rangle^{\uparrow} \mid b\langle y\rangle \diamond a[y]) \end{aligned}$$

defines a membrane around kell $a[P]$ that models fail-stop failures with the possibility of recovery.

Dynamic Binding. In a distributed programming model, it is important to provide both local and remote equivalent of libraries or services, because of the cost, safety, and security considerations that may apply. Thus, it should be possible to access identically named libraries or services (like a print service) at different sites. Dynamic binding refers to this possibility of binding names of resources (functions, libraries, services) to different entities, depending on the locations of processes accessing resources. It is interesting to note that not all distributed process calculi support dynamic binding. The Distributed Join calculus, for instance, does not support it since each definition is uniquely defined: every resource is permanently bound to a single locality. In the Kell calculus, just as with Ambient calculi and the Seal calculus, dynamic binding support is a consequence of the local semantics of communication. Thus, we have the following reductions, which illustrate that the outcome of an invocation of function `fun` depends on the location of $P \triangleq \mathtt{fun}\langle Q\rangle$:

$$a[(\mathtt{fun}\langle x\rangle \diamond F_a(x)) \mid P] \mid b[(\mathtt{fun}\langle x\rangle \diamond F_b(x)) \mid P] \;\rightarrow^*$$

$$a[(\mathtt{fun}\langle x\rangle \diamond F_a(x)) \mid F_a(Q)] \mid b[(\mathtt{fun}\langle x\rangle \diamond F_b(x)) \mid F_b(Q)]$$

3 The Kell Calculus: Syntax and Operational Semantics

3.1 Syntax

The syntax of the Kell calculus is given in Figure 1. It is parameterized by the pattern language used to define patterns ξ in triggers $\xi \triangleright P$.

Names and Variables. We assume an infinite set N of *names*, and an infinite set V of *process variables*. We assume that $\mathsf{N} \cap \mathsf{V} = \emptyset$. We let a, b, n, m and their decorated variants range over N; and p, q, x, y range over V. The set L of *identifiers* is defined as $\mathsf{L} = \mathsf{N} \cup \mathsf{V}$.

Processes. Terms in the Kell calculus grammar are called *processes*. We note $\mathsf{K}_{\mathcal{L}}$ the set of Kell calculus processes with patterns in pattern language $\mathcal{L}$. In most cases the pattern language used is obvious from the context, and we simply write K. We let P, Q, R, S, T

$$P ::= \mathbf{0} \;\mid\; x \;\mid\; \xi \triangleright P \;\mid\; \nu a.P \;\mid\; a\langle P\rangle.P \;\mid\; P \mid P \;\mid\; a\,[P]\,.P$$
$$a \in \mathsf{N}, \quad x \in \mathsf{V}$$

Fig. 1. Syntax of the Kell Calculus

and their decorated variants range over processes. We call *message* a process of the form $a\langle P\rangle.Q$. We let M, N and their decorated variants range over messages and parallel composition of messages. We call *kell* a process of the form $a\,[P]\,.Q$. The name a in a kell $a\,[P]\,.Q$ is called the name of the kell. In a kell of the form $a\,[\ldots \mid a_j\,[P_j] \mid \ldots]$ we call *subkells* the processes $a_j\,[P_j]$.

Abbreviations and Conventions. We abbreviate $a\langle P\rangle$ a message of the form $a\langle P\rangle.\mathbf{0}$. We abbreviate a a message of the form $a\langle\mathbf{0}\rangle$. We abbreviate $a\,[P]$ a kell of the form $a\,[P]\,.\mathbf{0}$. In a term $\nu a.P$, the scope extends as far to the right as possible. In a term $\xi \triangleright P$, the scope of $\triangleright$ extends as far to the left and to the right as possible. Thus, $a\langle c\rangle \mid b\,[y] \triangleright P \mid Q$ stands for $(a\langle c\rangle \mid b\,[y]) \triangleright (P \mid Q)$. We use standard abbreviations from the the π-calculus: $\nu a_1 \ldots a_q.P$ for $\nu a_1. \ldots \nu a_q.P$, or $\nu\widetilde{a}.P$ if $\widetilde{a} = (a_1 \ldots a_q)$. By convention, if the name vector $\widetilde{a}$ is null, then $\nu\widetilde{a}.P \triangleq P$. Also, we abuse notation and note $\widetilde{a}$ the set $\{a_1, \ldots, a_n\}$, where $\widetilde{a}$ is the vector $a_1 \ldots a_n$. We note $\prod_{j \in J} P_j$, $J = \{1, \ldots, n\}$ the parallel composition $(P_1 \mid (\ldots (P_{n-1} \mid P_n) \ldots))$. By convention, if $J = \emptyset$, then $\prod_{j \in J} P_j \triangleq \mathbf{0}$.

For the definition of the operational semantics of the calculus, we use additional terms called *annotated messages*. Annotated messages comprise:

- *Local Messages*: a local message is a term of the form $a\langle P\rangle$. We write M_m for a multiset of local messages.
- *Up Messages*: an up message is a term of the form $a\langle P\rangle^{\uparrow_b}$. We write M_u for a multiset of up messages.
- *Down Messages*: a down message is a term of the form $a\langle P\rangle^{\downarrow_b}$. We write M_d for a multiset of down messages.
- *Kell Messages*: a kell message is a term of the form $a\,[P]$. We write M_k for a multiset of kell messages.

We write M for a multiset of annotated messages. Some of these terms are not processes, namely those in M_u and M_d; they are only used for matching purposes. We often write these multisets as parallel compositions of annotated messages.

Let M_m be a multiset of local messages. We write $M_m^{\uparrow_b}$ for the multiset of up messages $\{m^{\uparrow_b} \mid m \in M_m\}$, and $M_m^{\downarrow_b}$ for the multiset of down messages $\{m^{\downarrow_b} \mid m \in M_m\}$.

Let $M = \{m_j^{n_j} \mid j \in J\}$ be an arbitrary multiset (where the multiplicity of element m_j is n_j). We note $M.\mathsf{supp} = \{m_j \mid j \in J\}$ the support set of M, i.e. the smallest set to which elements of M belong.

Contexts. A Kell calculus context is a term $\mathbf{C}$ built according to the grammar given in Figure 2. Filling the hole in $\mathbf{C}$ with a Kell calculus term Q results in a Kell calculus term noted $\mathbf{C}\{Q\}$. We note C the set of Kell calculus contexts. We let $\mathbf{C}$ and its decorated variants range over C. We also make use of a specific form of contexts, called execution contexts (noted $\mathbf{E}$), which are used to specify the operational semantics of the calculus.

$$\begin{aligned}
\mathbf{C} ::= {} & \cdot \;\mid\; \xi \triangleright \mathbf{C} \;\mid\; \nu a.\mathbf{C} \;\mid\; (P \mid \mathbf{C}) \;\mid\; a\,[\mathbf{C}]\,.P \\
& \mid\; a\langle \mathbf{C} \rangle.P \;\mid\; a\langle P \rangle.\mathbf{C} \;\mid\; a\,[P]\,.\mathbf{C} \\
\mathbf{E} ::= {} & \cdot \;\mid\; \nu a.\mathbf{E} \;\mid\; a\,[\mathbf{E}]\,.P \;\mid\; P \mid \mathbf{E}
\end{aligned}$$

Fig. 2. Syntax of Contexts

Substitutions. We call substitution a (partial) function $\theta : (\mathsf{N} \to \mathsf{N}) \uplus (\mathsf{V} \to \mathsf{K})$ from names to names and process variables to Kell calculus processes. We write $P\theta$ the image under the substitution θ of process P. We note Θ the set of substitutions.

If $f : \mathcal{E} \to \mathcal{F}$ is a partial function, we note $f(x) \downarrow$ to indicate that f is defined at $x \in \mathcal{E}$. We note $f(x) \uparrow$ for $\neg(f(x) \downarrow)$. We note $f.\mathtt{dom} = \{x \in \mathcal{E} \mid f(x) \downarrow\}$ the domain of function f and $f.\mathtt{ran} = \{y \in \mathcal{F} \mid \exists x \in \mathcal{E}, f(x) = y\}$ the range of function f.

We note $\perp$ the substitution that is defined nowhere, i.e. such that $\perp.\mathtt{dom} = \emptyset$.

The substitution $\theta \oplus \theta'$ is defined iff

- both θ and θ' are defined, i.e. if θ and θ' are both different from $\perp$;
- let $\mathtt{dom}_\mathtt{n}(\theta) = \theta.\mathtt{dom} \cap \mathsf{N}$ be the domain of the substitution θ restricted to names; for all $n \in \mathtt{dom}_\mathtt{n}(\theta) \cap \mathtt{dom}_\mathtt{n}(\theta')$, we have $\theta(n) = \theta'(n)$ (that is: if the name n is in the domain of both substitutions, it is mapped to the same name: one may test for name equality);
- let $\mathtt{dom}_\mathtt{P}(\theta) = \theta.\mathtt{dom} \cap \mathsf{V}$ be the domain of the substitution θ restricted to process variables; then $\mathtt{dom}_\mathtt{P}(\theta) \cap \mathtt{dom}_\mathtt{P}(\theta') = \emptyset$ (one may not test for process equality).

If defined, $\theta \oplus \theta'$ is the union of θ and θ'. We extend this operation to sets of substitutions:

$$S_1 \oplus S_2 \triangleq \{\theta_1 \oplus \theta_2 \;\mid\; \theta_1 \in S_1,\; \theta_2 \in S_2,\; \theta_1 \oplus \theta_2 \text{ defined}\}$$

Patterns. A pattern ξ is an element of a pattern language $\mathcal{L}$. A pattern ξ acts as a binder in the calculus. A pattern can bind name variables, of the form (a), where $a \in \mathsf{N}$, and process variables. All name and process variables appearing in a pattern ξ are bound by the pattern. Name variables can only match names. Process variables can only match processes. Patterns are supposed to be linear with respect to process variables, that is, each process variable x occurs only once in a given pattern ξ.

We make the following assumptions on a pattern language $\mathcal{L}$:

- A pattern language $\mathcal{L}$ is a set of patterns that are multisets of *single patterns* ξ_m, ξ_d, ξ_u, and ξ_k. The language $\mathcal{L}$ can be described by a grammar, with the multiset union being represented by parallel composition.
  - ξ_m is taken from the set Ξ_m and is a *local message pattern*: it is used to match local messages;
  - ξ_d is taken from the set Ξ_d and is a *down message pattern*: it is used to match messages from immediate subkells;
  - ξ_u is taken from the set Ξ_u and is a *up message pattern*: it is used to match messages from the environment of the enclosing kell;
  - ξ_k is taken from the set Ξ_k and is a *kell message pattern*: it is used to match immediate subkells.

$$\begin{array}{ll}
\mathtt{fn}(\mathbf{0}) = \emptyset & \mathtt{fv}(\mathbf{0}) = \emptyset \\
\mathtt{fn}(a) = \{a\} & \mathtt{fv}(a) = \emptyset \\
\mathtt{fn}(x) = \emptyset & \mathtt{fv}(x) = \{x\} \\
\mathtt{fn}(\nu a.P) = \mathtt{fn}(P) \setminus \{a\} & \mathtt{fv}(\nu a.P) = \mathtt{fv}(P) \\
\mathtt{fn}(a\,[Q]\,.P) = \mathtt{fn}(a, Q, P) & \mathtt{fv}(a\,[Q]\,.P) = \mathtt{fv}(Q, P) \\
\mathtt{fn}(a\langle P\rangle.Q) = \mathtt{fn}(a, P, Q) & \mathtt{fv}(a\langle P\rangle.Q) = \mathtt{fv}(P, Q) \\
\mathtt{fn}(P \mid Q) = \mathtt{fn}(P, Q) & \mathtt{fv}(P \mid Q) = \mathtt{fv}(P, Q) \\
\mathtt{fn}(\xi \triangleright P) = \mathtt{fn}(\xi) \cup (\mathtt{fn}(P) \setminus \mathtt{bn}(\xi)) & \mathtt{fv}(\xi \triangleright P) = \mathtt{fv}(P) \setminus \mathtt{bv}(\xi)
\end{array}$$

Fig. 3. Free names and free variables

- One can decide whether a pattern matches a given term. More precisely, each pattern language is equipped with a decidable relation match, which associates a pair $\langle \xi, M\rangle$, consisting of a pattern ξ and a multiset of annotated messages M, with defined substitutions that make the pattern match the multiset of annotated messages, if there are such substitutions, and with $\emptyset$ otherwise (see section 3.2 for more details). We write $\theta \in \mathtt{match}(\xi, M)$ for $\langle\langle \xi, M\rangle, \theta\rangle \in \mathtt{match}$.
- Pattern languages are equipped with three functions fn, bn, and bv, that map a pattern ξ to its set of free names, bound name variables, and bound process variables, respectively. Note that patterns may have free names, but cannot have free process variables (i.e. all process variables appearing in a pattern are bound in the pattern).
- Pattern languages are equipped with a function sk, which maps a pattern ξ to a multiset of names. Intuitively, $\xi.\mathtt{sk}$ corresponds to the multiset of channel names on which pattern ξ expects messages or kells (we use indifferently an infix or a postfix notation for sk). We identify $\xi.\mathtt{sk} = \{a_i \mid i \in I\}$ with the action $\prod_{i\in I} a_i$ (see section 3.3 for more details). By definition, we set $\xi.\mathtt{sk}.\mathtt{supp} \subseteq \mathtt{fn}(\xi)$. In other terms, a pattern may not bind a name that appears in its set of channel names (a trigger must know channel names in order to receive messages on them).
- Pattern languages are equipped with a structural congruence relation between patterns, noted $\equiv$. We assume the following properties: if $\xi \equiv \zeta$, then $\mathtt{fn}(\xi) = \mathtt{fn}(\zeta)$, $\xi.\mathtt{sk} = \zeta.\mathtt{sk}$, and $\mathtt{bn}(\xi) \cup \mathtt{bv}(\xi) = \mathtt{bn}(\zeta) \cup \mathtt{bv}(\zeta)$. Moreover, the interpretation of join patterns as multisets of simple patterns implies that the structural congruence on patterns must include the associativity and commutativity of the parallel composition operator.
- A pattern language is compatible with the structural congruence defined below (see section 3.2 for more details), in particular if $P \equiv Q$ then there is no Kell calculus context that can distinguish between P and Q.

Free Names and Free Variables. The other binder in the calculus is the ν operator, which corresponds to the restriction operator of the π-calculus. Notions of free names (fn) and free variables (fv) are classical and are defined in Figure 3. We note $\mathtt{fn}(P_1, \ldots, P_n)$ to mean $\mathtt{fn}(P_1) \cup \ldots \cup \mathtt{fn}(P_n)$, and likewise for other functions operating on free or bound identifiers. We note $P =_\alpha Q$ when two terms P and Q are α-convertible.

^{2}size

$$(P \mid Q) \mid R \equiv P \mid (Q \mid R) \text{ [S.PAR.A]} \qquad P \mid Q \equiv Q \mid P \text{ [S.PAR.C]}$$

$$P \mid \mathbf{0} \equiv P \text{ [S.PAR.N]} \qquad \nu c.\mathbf{0} \equiv \mathbf{0} \text{ [S.NU.NIL]} \qquad \nu a.\nu b.P \equiv \nu b.\nu a.P \text{ [S.NU.C]}$$

$$\frac{a \notin \mathtt{fn}(Q)}{(\nu a.P) \mid Q \equiv \nu a.P \mid Q} \text{ [S.NU.PAR]} \qquad \frac{a \notin \mathtt{fn}(b, Q)}{b[\nu a.P].Q \equiv \nu a.b[P].Q} \text{ [S.NU.KELL]}$$

$$\frac{\xi \equiv \zeta}{\xi \triangleright P \equiv \zeta \triangleright P} \text{ [S.TRIG]} \qquad \frac{P =_\alpha Q}{P \equiv Q} \text{ [S.}\alpha\text{]} \qquad \frac{P \equiv Q}{\mathbf{C}\{P\} \equiv \mathbf{C}\{Q\}} \text{ [S.CONTEXT]}$$

Fig. 4. Structural congruence

3.2 Reduction Semantics

We define in this section a reduction semantics for Kell calculus processes. As usual, we use a structural congruence relation, and a reduction step relation.

Structural Congruence. The structural congruence $\equiv$ is the smallest equivalence relation that verifies the rules in Figure 4. The rules S.PAR.A, S.PAR.C, S.PAR.N state that the parallel operator $\mid$ is associative, commutative, and has $\mathbf{0}$ as a neutral element. Note that, in rule S.TRIG, we rely on the structural congruence relation on patterns, also noted $\equiv$. Note also that we *do* allow the Ambient-like rule S.NU.KELL.

The *compatibility* of the relation `match` with the structural congruence is formally defined as follows.

Definition 1. *Two substitutions θ, θ' are said to be* equivalent *(noted $\theta \equiv \theta'$), if $\theta.\mathtt{dom} = \theta'.\mathtt{dom}$, and, for all $a, x \in \theta.\mathtt{dom}$, $a\theta = a\theta'$, and $x\theta \equiv x\theta'$.*

Definition 2. *A pattern language $\mathcal{L}$ is said to be* compatible *with the structural congruence $\equiv$ if, for all $\xi, \zeta \in \mathcal{L}$, and all multisets of annotated messages M, M', whenever $\xi \equiv \zeta$, $M \equiv M'$, if $\theta \in \mathtt{match}(\xi, M)$, then there exists $\theta' \in \mathtt{match}(\xi, M')$ such that $\theta \equiv \theta'$.*

Sub-reduction. We define a sub-reduction relation $\leadsto$: $\mathsf{K} \to \mathsf{K}$ mapping processes to processes. We note $\leadsto^*$ the reflexive and transitive closure of the sub-reduction relation.

The sub-reduction relation is defined to handle scope extrusion of restriction out of kell boundaries. Some explanation is in order. In presence of running process replication, if one is not careful, the use of the structural congruence rule S.NU.KELL could give rise to behaviour which would violate the idea that structurally equivalent processes should behave similarly in the same evaluation context. The example below provides an illustration:

$$\begin{aligned}(a[x] \triangleright x \mid x) \mid a[\nu b.P] &\;\to\; (\nu b.P) \mid (\nu b.P)\\ (a[x] \triangleright x \mid x) \mid \nu b.a[P] &\;\to\; \nu b.P \mid P\end{aligned}$$

A similar phenomenon arises with the Seal calculus. In the Seal calculus and in earlier versions of the Kell calculus [35], the issue is handled by suppressing rule S.NU.KELL from the structural congruence rules and by handling scope extrusion out of localities during reduction steps (what can be called "dynamic scope extrusion"), only when it is necessary (i.e. when a communication across a locality boundary needs to take place). We have found that dynamic scope extrusion gave rise to undue complexities in a virtual machine implementation of the calculus.

In [10], authors of the Seal calculus argue in favor of a specific form of dynamic scope extrusion: the movement of processes outside a given locality can only take place if all of their bound names have seen their scope extruded out of the enclosing locality during a prior communication. In other terms, not only does scope extrusion out of a locality only takes place during first-order communication (as in the previous case of dynamic scope extrusion), but the movement of processes out of a given locality only takes place if all of their bound names have been previously thus extruded. The argument behind this form of dynamic scope extrusion is that of security: names restricted inside a given locality (e.g. a name such as a in $b[\nu a.P]$) should be interpreted as local channels, which should only become visible outside their enclosing locality by explicit communication. Thus, a process which has non-extruded local channels is prevented from moving, so as to not unduly expose these local channels. This argument, however, is a bit moot. First, nothing prevents the unwary programmer from inadvertently turning a local channel into a global one by communicating it outside of the enclosing locality. If the distinction between local channels and global channels is important, we would expect some additional means, such as a type system, to enforce it. Second, note that in a form of dynamic scope extrusion where scope is systematically extended outside of a locality prior to the move of a process out of that locality, restricted names still remain private in absence of communication to other processes. A security issue therefore only arises when restricted names are communicated to the wrong recipient (e.g. one with the wrong security credentials), and the form of dynamic scope extrusion proposed in [10] does not prevent it from occurring.

In this paper, we therefore adopt the following simpler approach: Ambient-like scope extrusion is allowed in the structural congruence, but only processes *in normal form* can reduce. A process is in normal form if all restrictions under evaluation context have been pushed to the top-level. More formally, P is in normal form iff there is no P' such that $P \rightsquigarrow P'$. In our example above, $a[\nu b.P]$ is not in normal form, since one can apply rule rule SR.KELL to it, and thus $(a[x] \triangleright x \mid x) \mid a[\nu b.P]$ can reduce only after sub-reduction:

$$(a[x] \triangleright x \mid x) \mid a[\nu b.P] \rightsquigarrow^* \nu b.(a[x] \triangleright x \mid x) \mid a[P] \to \nu b.P \mid P$$

The sub-reduction relation is defined as the smallest relation satisfying the rules in Figure 5. One can note that $\rightsquigarrow^* \subseteq \equiv$, i.e. that for all P, Q, $P \rightsquigarrow^* Q$ implies that $P \equiv Q$.

Reduction. The reduction relation $\to$ is defined as the smallest binary relation on K^2 that satisfies the rules given in Figure 7. We define the predicates Δ, Υ, Ψ in Figure 6.

Predicate Δ (resp. Υ, Ψ) can be read as a function that, given a set U of local messages (resp. of kells, of messages in sub-kells), extracts a multiset M_m (resp. M_k, M_d) of local messages (resp. of kell messages, of down messages) to match, and a residual V. Residuals are continuation processes that appear after a reduction step. Note the condition $U_2 \not\rightsquigarrow$ in rules R.RED.L and R.RED.G: it prevents a reduction to take place

$$\frac{a \notin \mathtt{fn}(b, Q)}{b[\nu a.P].Q \rightsquigarrow \nu a.b[P].Q}\ [\text{SR.K}\textsc{ell}] \qquad \frac{a \notin \mathtt{fn}(P)}{\nu a.P \rightsquigarrow P}\ [\text{SR.GC}]$$

$$\frac{a \notin \mathtt{fn}(Q)}{(\nu a.P) \mid Q \rightsquigarrow \nu a.P \mid Q}\ [\text{SR.P}\textsc{ar}\text{.L}] \qquad \frac{a \notin \mathtt{fn}(Q)}{Q \mid (\nu a.P) \rightsquigarrow \nu a.Q \mid P}\ [\text{SR.P}\textsc{ar}\text{.R}]$$

$$\frac{P =_\alpha P' \qquad P' \rightsquigarrow Q}{P \rightsquigarrow Q}\ [\text{SR.}\alpha] \qquad \frac{P \rightsquigarrow Q}{\mathbf{E}\{P\} \rightsquigarrow \mathbf{E}\{Q\}}\ [\text{SR.C}\textsc{tx}]$$

Fig. 5. Sub-reduction relation

$$\Delta(U, M_m, V) \iff \begin{cases} U = \prod_{j \in J} a_j \langle P_j \rangle .Q_j \\ M_m = \prod_{j \in J} a_j \langle P_j \rangle \\ V = \prod_{j \in J} Q_j \end{cases}$$

$$\Upsilon(U, M_k, V) \iff \begin{cases} U = \prod_{j \in J} a_j\, [P_j]\, .Q_j \\ M_k = \prod_{j \in J} a_j\, [P_j] \\ V = \prod_{j \in J} Q_j \end{cases}$$

$$\Psi(U, M_d, V) \iff \begin{cases} U = \prod_{j \in J} a_j \left[R_j \mid \prod_{i \in I_j} a_i \langle P_j \rangle .S_i \right] .Q_j \\ M_d = \prod_{j \in J} \prod_{i \in I_j} a_i \langle P_i \rangle^{\downarrow a_j} \\ V = \prod_{j \in J} a_j \left[R_j \mid \prod_{i \in I_j} S_i \right] .Q_j \end{cases}$$

Fig. 6. Reduction predicates

with kell messages in non-normal form; this takes care of the issue of dynamic scope extrusion discussed earlier.

The reduction relation depends on a matching relation, `match`, which associates pairs consisting of a multiset of patterns drawn from $\mathcal{L}$ and a multiset of annotated messages M with substitutions (from names to names and from process variables to processes). This matching relation is assumed to be defined in terms of four functions, $\mathtt{match}_m$, $\mathtt{match}_d$, $\mathtt{match}_u$, and $\mathtt{match}_k$, that define how a single pattern matches a single annotated message. Each of these functions takes a single pattern ξ^r and an annotated message M, an returns a defined substitution θ is the annotated message matches the pattern, and the undefined substitution $\perp$ otherwise. Noting ξ^r a single pattern and $\Sigma(J)$ the set of permutations on finite set J, we have:

$$\dfrac{\begin{array}{c}\xi \neq \emptyset \qquad \theta \in \mathtt{match}(\xi, M_m \mid M_d \mid M_k) \\ \varDelta(U_1, M_m, V_1) \qquad \varUpsilon(U_2, M_k, V_2) \qquad \varPsi(U_3, M_d, V_3) \qquad U_2 \not\rightsquigarrow\end{array}}{(\xi \triangleright P) \mid U_1 \mid U_2 \mid U_3 \rightarrow P\theta \mid V_1 \mid V_2 \mid V_3}\ [\text{R.}\textsc{Red}\text{.L}]$$

$$\dfrac{P \equiv P' \qquad P' \rightarrow Q' \qquad Q' \equiv Q}{P \rightarrow Q}\ [\text{R.}\textsc{Eqv}] \qquad\qquad \dfrac{P \rightarrow Q}{\mathbf{E}\{P\} \rightarrow \mathbf{E}\{Q\}}\ [\text{R.}\textsc{Ctx}]$$

$$\dfrac{\begin{array}{c}\xi \neq \emptyset \qquad \theta \in \mathtt{match}(\xi, M_m \mid M_d \mid M_k \mid M^{\uparrow_b}) \qquad \varDelta(U_1, M_m, V_1) \\ \varUpsilon(U_2, M_k, V_2) \qquad \varPsi(U_3, M_d, V_3) \qquad \varDelta(U_4, M, V_4) \qquad U_2 \not\rightsquigarrow\end{array}}{b[(\xi \triangleright P) \mid U_1 \mid U_2 \mid U_3 \mid R].T \mid U_4 \rightarrow b[P\theta \mid V_1 \mid V_2 \mid V_3 \mid R].T \mid V_4}\ [\text{R.}\textsc{Red}\text{.G}]$$

Fig. 7. Reduction Relation

$$\begin{aligned}
\mathtt{match}(\prod_{j \in J} \xi_j^r, \prod_{j \in J} M_j) &= \bigcup_{\sigma \in \varSigma(J)} \left(\bigoplus_{j \in J} \mathtt{match}(\xi_j^r, M_{\sigma(j)}) \right) \\
\mathtt{match}(\xi_m, a\langle P \rangle) &= \mathtt{match}_m(\xi_m, a\langle P \rangle) \\
\mathtt{match}(\xi_d, a\langle P \rangle^{\downarrow_b}) &= \mathtt{match}_d(\xi_d, a\langle P \rangle^{\downarrow_b}) \\
\mathtt{match}(\xi_u, a\langle P \rangle^{\uparrow_b}) &= \mathtt{match}_u(\xi_u, a\langle P \rangle^{\uparrow_b}) \\
\mathtt{match}(\xi_k, a\,[P]) &= \mathtt{match}_k(\xi_k, a\,[P])
\end{aligned}$$

Example 1. To illustrate these definitions, we introduce a first instance of the Kell calculus with a simple pattern language. We call this calculus jK. The patterns in jK are defined by the following grammar:

$$\begin{array}{llll}
\xi ::= J \quad \mid \quad \xi_k \quad \mid \quad J \mid \xi_k & & J ::= \xi_m \quad \mid \quad \xi_d \quad \mid \quad \xi_u \quad \mid \quad J \mid J & \\
\xi_m ::= a\langle x \rangle & \xi_u ::= a\langle x \rangle^{\uparrow} & \xi_d ::= a\langle x \rangle^{\downarrow} & \xi_k ::= a\,[x]
\end{array}$$

The matching functions for jK patterns are defined inductively as follows:

$$\begin{array}{cc}
\mathtt{match}_m(a\langle x \rangle, a\langle P \rangle) \triangleq \{{}^{P}\!/_{x}\} & \mathtt{match}_d(a\langle x \rangle^{\downarrow}, a\langle P \rangle^{\downarrow_b}) \triangleq \{{}^{P}\!/_{x}\} \\
\mathtt{match}_u(a\langle x \rangle^{\uparrow}, a\langle P \rangle^{\uparrow_b}) \triangleq \{{}^{P}\!/_{x}\} & \mathtt{match}_k(a\,[x]\,, a\,[P]) \triangleq \{{}^{P}\!/_{x}\}
\end{array}$$

Note that, apart from the use of join patterns (i.e. the possibility to receive multiple messages at once), the pattern language of jK is extremely simple and does not allow for name-passing.

The $\mathtt{sk}$ function for jK patterns is defined as follows:

$$\begin{gathered}
(a\,[x]).\mathtt{sk} = (a\langle x \rangle).\mathtt{sk} = a\langle x \rangle^{\downarrow} = a\langle x \rangle^{\uparrow} = a \\
(\xi_1 \mid \xi_2).\mathtt{sk} = \xi_1.\mathtt{sk} \mid \xi_2.\mathtt{sk}
\end{gathered}$$

Example 2. The reduction rules R.RED.L and R.RED.G appear formidable, but only because they allow arbitrary combination of local, up, down and kell messages to be received by a trigger. Using simple jK patterns, on can see immediately that the following rules are derived reduction rules in jK:

$$(a\langle x\rangle \triangleright P) \mid a\langle Q\rangle.S \rightarrow P\{{}^{Q}\!/_{x}\} \mid S \text{ [Local]}$$

$$(a\langle x\rangle^{\downarrow} \triangleright P) \mid b[a\langle Q\rangle.S \mid R].T \rightarrow P\{{}^{Q}\!/_{x}\} \mid b[S \mid R].T \text{ [Out]}$$

$$b[(a\langle x\rangle^{\uparrow} \triangleright P) \mid R].T \mid a\langle Q\rangle.S \rightarrow b[P\{{}^{Q}\!/_{x}\} \mid R].T \mid S \text{ [In]}$$

$$(a\,[x] \triangleright P) \mid a\,[Q]\,.S \rightarrow P\{{}^{Q}\!/_{x}\} \mid S \text{ [Kell]}$$

One can notice that the rules Local, Out, In, and Kell correspond to the four kinds of actions discussed in Section 2.

3.3 Labelled Transition System Semantics

We define in this section a labelled transition system for Kell calculus processes. The labelled transition system is defined by means of the same sub-reduction relation as for the reduction semantics, and of a commitment relation in the style of the commitment rules for the π-calculus defined in [27].

Abstractions and Concretions. We define the reduction relation using an extension of Milner's concretions and abstractions [27]. Extensions are required to the original notions because of two features, which are peculiar to the Kell calculus input patterns:

1. The possibility for patterns to involve the simultaneous receipt of multiple messages (as for Join patterns), as illustrated with jK. This is the reason for the presence of the name a in an abstraction $(a\langle x\rangle)P$.
2. The possibility for patterns to rely on contextual information (enabling, for instance, to capture the origin of messages, as indicated by the $^{\uparrow}$ and $^{\downarrow}$ arrows in jK patterns).

We first define *concretions*. A concretion C, D consists of a multiset of annotated messages (which are emitted), and a process that represents the rest of the computation. Their syntax is given by the grammar in Figure 8, where P is as in Figure 1. The Ω production of the grammar in Figure 8 yields (possibly empty) multisets of annotated messages, which do not contain any up message as labelled transitions do not create them. Up messages are dealt with in the definition of the pseudo-application operator below.

$$\begin{aligned} C &::= \nu\widetilde{a}.\Omega \parallel P \\ \Omega &::= \emptyset \quad | \quad a\langle P\rangle \quad | \quad a\langle P\rangle^{\downarrow_b} \quad | \quad a\,[P] \quad | \quad \Omega \mid \Omega \end{aligned}$$

Fig. 8. Syntax of Concretions

We then define *abstractions*. An abstraction F is given by the grammar in Figure 9, where ξ, P are as in Figure 1, and C is a concretion. We call *simple abstraction* an abstraction given by the G production in Figure 9.

$$\begin{aligned}F &::= G \quad|\quad a[G].P \quad|\quad F@C \quad|\quad \nu\widetilde{a}.F\\ G &::= (\xi)P \quad|\quad G@C\end{aligned}$$

Fig. 9. Syntax of Abstractions

Actions. Actions are given by the grammar in Figure 10, where $a \in \mathsf{N}$, ϵ and τ are two distinct symbols that do not belong to N. Classically, τ represents the silent action. Action ϵ is introduced merely for technical purposes, to signal the complete match of messages with an input pattern in a trigger. We denote Λ the set of actions, and we let α, β and their decorated variant range over Λ. By definition, the parallel operator $\mid$ on actions is associative and commutative, and has ϵ as a neutral element. Furthermore, we set $\overline{a} \mid a = \epsilon$. The set of free names of an action α is defined inductively by: $\mathtt{fn}(\epsilon) = \mathtt{fn}(\tau) = \emptyset$, $\mathtt{fn}(\overline{a}) = \{a\}$, $\mathtt{fn}(\alpha \mid \beta) = \mathtt{fn}(\alpha, \beta)$. Abusing the notation, we identify when necessary an action α with the set of names that occur in α. Thus, action $\alpha = a \mid b \mid b \mid \overline{c}$ is identified with the set $\{a, b, c\}$.

$$\alpha ::= \epsilon \quad|\quad \tau \quad|\quad a \quad|\quad \overline{a} \quad|\quad \alpha \mid \alpha$$

Fig. 10. Syntax of Actions

Agents. An agent A is a Kell calculus process P, a concretion C or an abstraction F. We note A the set of agents. We let A, B and their decorated variants range over agents; F and its decorated variants range over abstractions; G and its decorated variants range over simple abstractions; C, D and their decorated variants range over concretions.

The notions of free names, free variables, bound names and bound variables extend immediately to agents, noting that $\mathtt{fn}(a\langle P\rangle^{\downarrow_b}) = \mathtt{fn}(a, b, P)$, $\mathtt{fv}(a\langle P\rangle^{\downarrow_b}) = \mathtt{fv}(P)$, $\mathtt{fn}(\Omega \parallel P) = \mathtt{fn}(\Omega, P)$, $\mathtt{fv}(\Omega \parallel P) = \mathtt{fv}(\Omega, P)$, $\mathtt{fn}(F@C) = \mathtt{fn}(F, C)$ if $F@C \downarrow$, and $\mathtt{fv}(F@C = \mathtt{fv}(F, C)$ if $F@C \downarrow$ (see below the definition of the pseudo-application operator $@$).

Parallel Composition of Agents. We define the effect of parallel composition on agents, $\mid$: $\mathsf{A} \times \mathsf{A} \to \mathsf{A}$, as follows, where $C = \nu\widetilde{a}.\Omega \parallel P$ and $C' = \nu\widetilde{c}.\Omega' \parallel P'$:

$$\begin{aligned}&F \mid Q \triangleq F@(\emptyset \parallel Q)\\ &C \mid Q \triangleq \nu\widetilde{a}.\Omega \parallel (P \mid Q) \quad \text{if } \widetilde{a} \cap \mathtt{fn}(Q) = \emptyset\\ &C \mid C' \triangleq \nu\widetilde{a}.\nu\widetilde{c}.(\Omega \mid \Omega') \parallel (P \mid P') \quad \text{if } \widetilde{a} \cap \mathtt{fn}(\Omega', P') = \widetilde{c} \cap \mathtt{fn}(\Omega, P) = \emptyset\end{aligned}$$

For all annotated multisets of messages Ω, we set by definition $\Omega \mid \emptyset = \emptyset \mid \Omega = \Omega$.

Pseudo-Application. We define the pseudo-application relation, $@$, which in turns relies on the $\mathtt{match}$ relation. The relation $@$ gives the possible results of a successful reduction of an application. As a relation $@ : \mathsf{A} \times \mathsf{A} \times \mathsf{A}$, operation $@$ is partial and it is defined (written $\downarrow$) only when the last element B in a triple $\langle A_1, A_2, B\rangle \in @$ is a process. It is defined as follows (and undefined in all other cases), where we note $A_1@A_2 = B$ for $\langle A_1, A_2, B\rangle \in @$:

$$
\begin{gathered}
(F@C)@C' \triangleq F@(C \mid C') \\
(\xi)R@(\Omega \parallel P) \triangleq R\theta \mid P \quad \text{if } \theta \in \mathtt{match}(\xi, \Omega) \\
(a[(\xi)R@(\Omega \parallel P)].T)@(M_m \parallel Q) \triangleq a[R\theta \mid P].T \mid Q \quad \text{if } \theta \in \mathtt{match}(\xi, \Omega \mid M_m^{\uparrow_a}) \\
(\nu\widetilde{a}.F)@(\nu\widetilde{b}.C) \triangleq \nu\widetilde{a}.\nu\widetilde{b}.F@C \quad \text{if } \widetilde{a} \cap \mathtt{fn}(C) = \widetilde{b} \cap \mathtt{fn}(F) = \emptyset
\end{gathered}
$$

Note that in the third clause of the above definition, we can have $\Omega = \emptyset$, which accounts for the possibility of triggers only receiving up messages. In this case the above clause reduces to:

$$(a[(\xi)R@(\emptyset \parallel P)].T)@(M_m \parallel Q) \triangleq a[R\theta \mid P].T \mid Q \text{ if } \mathtt{match}(\xi, M_m^{\uparrow_a}) \ni \theta$$

We write $F@C\downarrow$ to indicate that the partial relation $@ : \mathsf{A} \times \mathsf{A} \times \mathsf{K}$ is defined on the pair $\langle F, C\rangle$ (i.e. that there exists $P \in \mathsf{K}$ such that $F@C = P$), and $F@C\uparrow$ to indicate that it is not.

Commitment Relation. The commitment relation is the smallest relation $\mathcal{R} \subseteq \mathsf{K} \times \Lambda \times \mathsf{A}$, that satisfies the rules in Figure 11.

A few comments are in order. First, one can note that actions in the commitment relation provide relatively few information on the nature of the operation they signal. For instance, $a\langle P\rangle.Q$, $a[P].Q$, $b[a\langle P\rangle.Q \mid R].S$ all give rise to a transition $\xrightarrow{\overline{a}}$. This is not a problem, however, for the concretions they give rise to, namely $a\langle P\rangle \parallel Q$, $a[P] \parallel Q$, $a\langle P\rangle^{\downarrow_a} \parallel a[Q \mid R].S$, a priori match different patterns. For instance, in jK, the patterns $a\langle x\rangle$, $a[x]$, and $a\langle x\rangle^{\downarrow}$ can distinguish between these different concretions. Second, communication involving multiple messages is dealt with by assembling abstractions and concretions in a piece-wise manner, until a match is found, which is signalled by a transition of the form $P \xrightarrow{\epsilon} Q$: action ϵ signals that all communication channels have been matched, and the presence of Q as the result of the transition signals that the match has been successful. Rule T.Red can then be used to effect the resulting silent transition. Third, note the presence of the conditions $a[P].Q \not\rightarrow$, $a[P].R \not\rightarrow$, $Q \mid P \not\rightarrow$, and $P \mid Q \not\rightarrow$ in rules T.Kell, T.Kell.P, T.Kell.C, T.Kell.F, T.Par.L, T.Par.R, T.Par.FC, T.Par.CF, T.Par.CC. This means that these rules operate only on processes in normal form. For processes which are not in normal form, rule T.SR must first be applied. This takes care, in the labelled transition semantics, of the dynamic scope extrusion issue discussed earlier. It also facilitates the proofs by induction, since one can essentially reason on processes in normal form only. Fourth, note the condition $\alpha \neq \epsilon$ in rules T.New, T.Par.L, T.Par.FC, T.Par.CF. Strictly speaking, this condition is not necessary, but it does simplify the handling of the different cases in the proofs by induction on the derivation of a transition $P \xrightarrow{\alpha} Q$.

The correspondence between the reduction semantics and the labelled transition semantics is given by the following theorem.

Theorem 1. *For all P, Q, $P \xrightarrow{\tau} \equiv Q$ iff $P \rightarrow Q$.*

$$a\langle P\rangle.Q \xrightarrow{\overline{a}} a\langle P\rangle \parallel Q \ [\text{T.}\textsc{Mess}] \qquad \frac{a[P].Q \not\rightsquigarrow}{a\,[P]\,.Q \xrightarrow{\overline{a}} a\,[P] \parallel Q}\ [\text{T.}\textsc{Kell}]$$

$$\xi \triangleright P \xrightarrow{\xi.\mathtt{sk}} (\xi)P \ [\text{T.}\textsc{Trig}] \qquad \frac{P \xrightarrow{\alpha} A \qquad a \notin \mathtt{fn}(\alpha) \qquad \alpha \neq \epsilon}{\nu a.P \xrightarrow{\alpha} \nu a.A}\ [\text{T.}\textsc{New}]$$

$$\frac{P \xrightarrow{\tau} Q \qquad a[P].R \not\rightsquigarrow}{a\,[P]\,.R \xrightarrow{\tau} a\,[Q]\,.R}\ [\text{T.}\textsc{Kell.P}] \qquad \frac{P \xrightarrow{\alpha} M_m \parallel Q \qquad a[P].R \not\rightsquigarrow}{a\,[P]\,.R \xrightarrow{\alpha} M_m^{\downarrow a} \parallel a\,[Q]\,.R}\ [\text{T.}\textsc{Kell.C}]$$

$$\frac{P \xrightarrow{\alpha} G \qquad a[P].R \not\rightsquigarrow}{a\,[P]\,.R \xrightarrow{\alpha} a\,[G]\,.R}\ [\text{T.}\textsc{Kell.F}] \qquad \frac{P \xrightarrow{\alpha} A \qquad \alpha \neq \epsilon \qquad P \mid Q \not\rightsquigarrow}{P \mid Q \xrightarrow{\alpha} A \mid Q}\ [\text{T.}\textsc{Par.L}]$$

$$\frac{P \xrightarrow{\alpha} A \qquad \alpha \neq \epsilon \qquad Q \mid P \not\rightsquigarrow}{Q \mid P \xrightarrow{\alpha} A \mid Q}\ [\text{T.}\textsc{Par.R}]$$

$$\frac{P \xrightarrow{\alpha} F \qquad Q \xrightarrow{\beta} C \qquad \alpha \neq \epsilon \qquad P \mid Q \not\rightsquigarrow}{P \mid Q \xrightarrow{\alpha \mid \beta} F@C}\ [\text{T.}\textsc{Par.FC}]$$

$$\frac{P \xrightarrow{\alpha} F \qquad Q \xrightarrow{\beta} C \qquad \alpha \neq \epsilon \qquad Q \mid P \not\rightsquigarrow}{Q \mid P \xrightarrow{\alpha \mid \beta} F@C}\ [\text{T.}\textsc{Par.CF}]$$

$$\frac{P \xrightarrow{\alpha} C \qquad Q \xrightarrow{\beta} C' \qquad P \mid Q \not\rightsquigarrow}{P \mid Q \xrightarrow{\alpha \mid \beta} C \mid C'}\ [\text{T.}\textsc{Par.CC}] \qquad \frac{P \xrightarrow{\epsilon} Q}{P \xrightarrow{\tau} Q}\ [\text{T.}\textsc{Red}]$$

$$\frac{P \rightsquigarrow^{*} Q \qquad Q \xrightarrow{\alpha} A}{P \xrightarrow{\alpha} A}\ [\text{T.SR}] \qquad \frac{Q =_{\alpha} P \qquad P \xrightarrow{\alpha} A \qquad A =_{\alpha} B}{Q \xrightarrow{\alpha} B}\ [\text{T.}\alpha]$$

Fig. 11. Commitment Relation

Proof. (Sketch[3]) The proof is entirely classical. It relies on the following result, which involves an extension of the structural congruence relation to agents: for all P, Q, when-

[3] The full proofs of the results reported in this paper can be found in the research report of the same title, available at `http : //sardes.inrialpes.fr/papers`.

ever $P \equiv Q$, if $P \xrightarrow{\alpha} A$, then there exists $B \equiv A$ such that $Q \xrightarrow{\alpha} B$, which is proved by induction on the derivation of $P \equiv Q$. □

4 Congruences for the Kell Calculus

4.1 Context Bisimulation

We first define a notion of context bisimulation for the Kell calculus, which is directly inspired by Sangiorgi's context bisimulation for HOπ [31]. We consider an early form of context bisimilarity. This is required to obtain a co-inductive characterization of barbed congruence for a large enough class of pattern languages (and in particular languages that support name passing).

We say that an agent A is *closed* if it contains no free process variable ($\mathtt{fv}(A) = \emptyset$). We note A_c and K_c the set of closed agents and closed processes, respectively. We use the (partial) operator $[\![\ ]\!]$ on agents, defined by:

$$a[\![\nu c.C]\!] \triangleq \nu c.a[\![C]\!] \text{ if } c \neq a \qquad a[\![\nu c.F]\!] \triangleq \nu c.a[\![F]\!] \text{ if } c \neq a$$
$$a[\![M_m \parallel T]\!].S \triangleq M_m^{\downarrow_a} \parallel a[T].S$$

Definition 3. *A relation $\mathcal{R} \subseteq \mathsf{K}_c^2$ is a* strong context simulation *on closed processes if for all P, Q closed, whenever $\langle P, Q\rangle \in \mathcal{R}$ we have:*

1. *If $P \xrightarrow{\tau} P'$, then there exists Q' such that $Q \xrightarrow{\tau} Q'$ and $\langle P', Q'\rangle \in \mathcal{R}$.*
2. *If $P \xrightarrow{\alpha} F$, then:*
   (a) *For all closed concretions C such that $F@C \downarrow$, there exists F' such that*
   $$Q \xrightarrow{\alpha} F', \text{ and } \langle F@C, F'@C\rangle \in \mathcal{R}.$$
   (b) *For all closed concretions C, D, and all a, T closed such that $(a[\![F@C]\!].T)@D \downarrow$, there exists F' such that $Q \xrightarrow{\alpha} F'$, and*
   $$\langle (a[\![F@C]\!].T)@D, (a[\![F'@C]\!].T)@D\rangle \in \mathcal{R}$$
3. *If $P \xrightarrow{\alpha} C$, then:*
   (a) *For all closed abstractions F such that $F@C \downarrow$, there exists C' such that $Q \xrightarrow{\alpha} C'$, and $\langle F@C, F@C'\rangle \in \mathcal{R}$.*
   (b) *For all closed abstractions F, all closed concretions D, and all a, T closed such that $(a[\![F@C]\!].T)@D \downarrow$, there exists C' such that $Q \xrightarrow{\alpha} C'$, and*
   $$\langle (a[\![F@C]\!].T)@D, (a[\![F@C']\!].T)@D\rangle \in \mathcal{R}$$
   (c) *For all closed abstractions F and all a, T closed such that $F@(a[\![C]\!].T) \downarrow$, there exists C' such that $Q \xrightarrow{\alpha} C'$, and*
   $$\langle F@(a[\![C]\!].T), F@(a[\![C']\!].T)\rangle \in \mathcal{R}$$

(d) For all closed abstractions F, all closed concretions D and all a, b, S, T closed such that $(a[\![F@(b[\![C]\!].S)]\!].T)@D\downarrow$, there exists C' such that $Q \xrightarrow{\alpha} C'$, and

$$\langle (a[\![F@(b[\![C]\!].S)]\!].T)@D, (a[\![F@(b[\![C']\!].S)]\!].T)@D\rangle \in \mathcal{R}$$

In the above definition, in the second clause concerning abstractions, note that quantification over concretions C includes concretions of the form $\emptyset \parallel \mathbf{0}$. This covers the cases $(a[F].T)@D\downarrow$.

Note also that the clauses in the above definition put no constraint on transitions of the form $P \xrightarrow{\epsilon} Q$, $P \xrightarrow{\alpha} F$ with F such that cases 2a and 2b are vacuously satisfied (there is no concretion that matches the abstraction), and $P \xrightarrow{\alpha} C$ with C such that cases 3a, 3b, 3c, and 3d are vacuously satisfied. Indeed, we deem such transitions to be irrelevant for comparing process behaviors. They are just artefacts of the commitment relation and do not constitute meaningful behavior: the first one is just an intermediate step in the derivation of the meaningful transition $P \xrightarrow{\tau} Q$, the other two are transitions resulting from ill-matched abstractions and concretions and are therefore not relevant.

The definition of strong context simulation can be motivated as follows. Intuitively, process Q simulates P if Q can match the silent moves of P (clause 1 of Definition 3), and if Q can match an action of P that transforms it into an abstraction (resp. a concretion) with an action that transforms it into an abstraction (resp. a concretion) that can simulates P's abstraction (resp. concretion) when applied to arbitrary concretions (resp. abstractions). When considering abstractions, for instance, one need to consider the different forms of application that may occur. It can be a direct application (clause 2a of Definition 3) or an application involving a kell context, e.g. because the abstractions considered require matching with an up message (clause 2b of Definition 3).

We can now define the notion of strong context bisimulation and the notion of strong context congruence. Notice that the definition of strong context congruence requires closure under substitution: this is because, as in the π-calculus, strong context bisimulation is not a congruence for input constructs.

Definition 4. *A relation $\mathcal{R} \subseteq \mathsf{K}_c^2$ is a* strong context bisimulation *on closed processes if $\mathcal{R}$ and its inverse $\mathcal{R}^{-1}$ are both strong context simulations.*

Two closed processes P and Q are said to be strongly context bisimilar*, noted $P \sim Q$, if there exists a strong context bisimulation $\mathcal{R}$ such that $\langle P, Q\rangle \in \mathcal{R}$.*

Two processes P and Q are said to be strongly context congruent*, noted $P \sim^c Q$, if for all substitutions θ such that $(\mathtt{fn}(P,Q) \cup \mathtt{fv}(P,Q)) \subseteq \theta.\mathtt{dom}$ and $(\theta.\mathtt{ran} \cap \mathsf{K}) \subseteq \mathsf{K}_c$, we have $P\theta \sim Q\theta$.*

Example 3. This example deals with the notion of firewall in the jK calculus. For any $P = e\langle T\rangle$, and any $Q = e\langle x\rangle^{\uparrow} \triangleright S$, we have $\nu a.a[b[P]] \sim \mathbf{0} \sim \nu a.a[b[Q]]$, for there are no transitions possible from any of those terms. In fact, in the Kell calculus, no interaction is possible between a process of the form $P = \nu a.a[b[U]]$ and its environment (although $P \rightarrow \nu a.a[b[U']]$ if $U \rightarrow U'$). The execution context $\mathbf{E} = \nu a.a[b[\cdot]]$ constitutes a *perfect firewall.*

The following proposition is classical.

Proposition 1. *The identity relation on* K^2 *is a strong context bisimulation. If* $\mathcal{R}_1$ *and* $\mathcal{R}_2$ *are strong context bisimulations, then so is their composition* $\mathcal{R}_1\mathcal{R}_2$*. The relation* $\sim$ *is an equivalence relation, and is the largest strong context bisimulation.*

We now define a sufficient condition on pattern languages to ensure that strong context congruence is indeed a congruence on Kell calculus processes. This condition is called *substitution-compatibility*. We first give an auxiliary definition.

Definition 5. *Let* θ, θ' *be two substitutions such that* $\theta.\mathtt{dom} = \theta'.\mathtt{dom}$*, and* $\mathcal{R}$ *be a binary relation on processes. We define* $\theta \; \mathcal{R} \; \theta'$ *as:* $\forall y \in (\theta.\mathtt{dom} \cap \mathsf{V}),\; y\theta \; \mathcal{R} \; y\theta'$.

Let $\mathcal{R}$ be a binary relation on processes. We extend $\mathcal{R}$ to vectors of processes thus: let $\widetilde{R}$ and $\widetilde{S}$ be two vectors of processes of the same size l, then $\widetilde{R} \; \mathcal{R} \; \widetilde{S}$ iff for all i, $1 \leq i \leq l,\; R_i \; \mathcal{R} \; S_i$.

Definition 6. *Let* $\mathcal{R} = \left\{ \langle P\{\widetilde{R}/\widetilde{x}\}, P\{\widetilde{S}/\widetilde{x}\}\rangle \mid P \in \mathsf{K}, \mathtt{fv}(P) \subseteq \widetilde{x}, \widetilde{R} \sim \widetilde{S} \right\}$*. A pattern language* $\mathcal{L}$ *is* substitution-compatible *iff for all* $\xi \in \mathcal{L}$*, and all* $M, \widetilde{R}, \widetilde{S}, \widetilde{x}$*, such that* $\widetilde{R}, \widetilde{S}$ *are closed,* $\widetilde{R} \sim \widetilde{S}$*, and* $\mathtt{fv}(M) \subseteq \widetilde{x}$*, we have:*

$$\mathtt{match}(\xi, M\{\widetilde{R}/\widetilde{x}\}) \ni \theta \quad \textit{implies} \quad \exists \theta', \mathtt{match}(\xi, M\{\widetilde{S}/\widetilde{x}\}) \ni \theta' \quad \textit{and} \quad \theta\mathcal{R}\theta'$$

Example 4. The pattern language of the jK calculus is substitution-compatible. Consider $\xi = a[x]$, then we have $\mathtt{match}(\xi, M) \neq \bot$ if $M = a[P]$. We have $M\{\widetilde{R}/\widetilde{y}\} = a[P\{\widetilde{R}/\widetilde{y}\}]$ and $M\{\widetilde{S}/\widetilde{y}\} = a[P\{\widetilde{S}/\widetilde{y}\}]$. Now, define

$$\theta \triangleq \mathtt{match}(\xi, M\{\widetilde{R}/\widetilde{y}\}) = \left\{ {}^{P\{^{\widetilde{R}}/_{\widetilde{y}}\}}/_{x} \right\}$$
$$\theta' \triangleq \mathtt{match}(\xi, M\{\widetilde{S}/\widetilde{y}\}) = \left\{ {}^{P\{^{\widetilde{S}}/_{\widetilde{y}}\}}/_{x} \right\}$$

If $\widetilde{R} \sim \widetilde{S}$, we have $x\theta \; \mathcal{R} \; x\theta'$, as required. The same is true for other patterns in jK.

Theorem 2. *If the pattern language* $\mathcal{L}$ *is substitution-compatible, then* $\sim^c$ *is a congruence on* K.

Proof. (Sketch) The crucial step in the proof is a substitution lemma which asserts that, if $R_1, \ldots, R_n$ and $S_1, \ldots S_n$ are closed processes such that $R_i \sim S_i$ for all $i \in \{1, \ldots, n\}$, then for all P such that $\mathtt{fv}(P) \subseteq \widetilde{x}$, we have $P\{\widetilde{R}/\widetilde{x}\} \sim P\{\widetilde{S}/\widetilde{x}\}$. The substitution lemma is proved by proving that the reflexive and transivitive closure $\mathcal{U}$ of the relation $\equiv \mathcal{R} \equiv$ is a strong context bisimulation, where $\mathcal{R} = \{\langle P, Q\rangle \mid P = \nu\widetilde{a}.H\{\widetilde{R}/\widetilde{x}\},\; Q = \nu\widetilde{b}.H\{\widetilde{S}/\widetilde{x}\},\; \widetilde{R} = R_1, \ldots, R_n,\; \widetilde{S} = S_1, \ldots, S_n,\; \mathtt{fv}(H) \subseteq \widetilde{x},\; \nu\widetilde{a}.R_i \sim \nu\widetilde{b}.S_i,\; \widetilde{a} \cap \mathtt{fn}(H) = \widetilde{b} \cap \mathtt{fn}(H) = \emptyset\}$. This is proved in turn by showing that $\mathcal{R}$ progresses to $\mathcal{U}$. The notion of progress is an adaptation to the Kell calculus context bisimulation of the notion of strong progress defined in chapter 2 of [32]. In turn, this is proved by induction on the derivation of the transition $P \xrightarrow{\alpha} A$, when considering $\langle P, Q\rangle \in \mathcal{R}$. □

It is worth pointing out that the technique used by Sangiorgi for the higher-order π-calculus [31] is not applicable in our context. Indeed, following the proof of the equivalent lemma in [31], would require us first to prove directly that $a[R] \sim a[S]$ if $R \sim S$, for R and S closed. But proving that $a[R] \sim a[S]$ implies in our case that we prove in particular that, for all P, $(a[x])P@a[R] \sim (a[x])P@a[S]$, i.e. that $P\{R/x\} \sim P\{S/x\}$, which leads to a circular argument.

4.2 Contextual Equivalence

We now define strong barbed bisimulation for the Kell calculus. Barbs of a Kell calculus process are defined as follows.

Definition 7. *Observability predicates, $\downarrow_a$, are defined as follows: $P \downarrow_a$ if one of the following cases holds:*

1. $P \equiv \nu\widetilde{c}.a\langle P'\rangle.Q \mid R$*, with* $a \notin \widetilde{c}$.
2. $P \equiv \nu\widetilde{c}.b[a\langle P'\rangle.Q \mid R].T \mid S$*, with* $a \notin \widetilde{c}$.
3. $P \equiv \nu\widetilde{c}.a\,[Q]\,.T \mid R$*, with* $a \notin \widetilde{c}$.

Intuitively, a barb on a signals a local message (clause 1 of Definition 7), a down message (clause 2 of Definition 7), or a kell message (clause 3 of Definition 7) on channel a. These observations are similar to those found e.g. in Ambient calculi. They are also relatively weak: for instance, they do not distinguish between a local message or a down message. However, they are valid observations, regardless of the pattern language used. One could of course imagine strenghening such observations, given more information on the pattern language. This is left for further study.

Definition 8. *A relation $\mathcal{R} \subseteq \mathsf{K}^2$ is a* strong barbed simulation *if whenever $\langle P, Q\rangle \in \mathcal{R}$, we have:*

1. *If $P \downarrow_a$ then $Q \downarrow_a$.*
2. *If $P \rightarrow P'$, then there exists Q' such that $Q \rightarrow Q'$ and $\langle P', Q'\rangle \in \mathcal{R}$.*

A relation $\mathcal{R}$ is a strong barbed bisimulation *if $\mathcal{R}$ and its inverse $\mathcal{R}^{-1}$ are both strong barbed simulations.*

Definition 9. *Two processes P and Q are said to be* strongly barbed bisimilar, *noted $P \sim_b Q$, if there exists a strong barbed bisimulation $\mathcal{R}$ such that $\langle P, Q\rangle \in \mathcal{R}$.*

Definition 10. *Two processes P and Q are said to be strong barbed congruent, noted $P \sim_b^c Q$, if, for all contexts $\mathbf{C}$ we have $\mathbf{C}\{P\} \sim_b \mathbf{C}\{Q\}$.*

The following proposition is classical.

Proposition 2. *$\sim_b$ is an equivalence relation and the largest strong barbed bisimulation. $\sim_b^c$ is an equivalence relation and the largest congruence included in $\sim_b$.*

For pattern languages that are substitution-compatible and which contain the pattern language of the jK calculus, the notion of strong barbed congruence and strong context congruence coincide. This is given by the following theorem:

Theorem 3. *If $\mathcal{L}$ is substitution-compatible and contains the pattern language of the* jK *calculus, then $\sim^c$ and $\sim^c_b$ coincide, i.e. for all $P, Q \in \mathsf{K}$, $P \sim^c Q$ iff $P \sim^c_b Q$.*

Proof. (Sketch) We first note that $\sim^c \subseteq \sim^c_b$, since $\sim \subseteq \sim_b$ and $\sim^c_b$ is the largest congruence included in $\sim_b$. We then stratify the $\sim$ relation, i.e. we define a family of relations $\sim_k$, where k ranges over integers, such that $\sim = \bigcap \sim_k$. We then prove by induction that for all integers k, if $P \not\sim_k Q$, then there exists a context $\mathbf{C}$ such that $\mathbf{C}\{P\} \not\sim_b \mathbf{C}\{Q\}$.

The induction case is dealt with by considering $P \xrightarrow{\alpha} A$ and the different cases for α, exhibiting appropriate discriminating contexts for each case. □

5 Instantiating the Kell Calculus

We now present several instantiations of the pattern language for the Kell calculus that illustrate the flexibility of our parameterized approach.

5.1 A Polyadic Name-Passing jK

$$\xi ::= J \quad | \quad \xi_k \quad | \quad J \mid \xi_k \qquad J ::= \xi_m \quad | \quad \xi_d \quad | \quad \xi_u \quad | \quad J \mid J$$
$$\xi_m ::= a\langle\overline{\rho}\rangle \qquad \xi_u ::= a\langle\overline{\rho}\rangle^{\uparrow} \qquad \xi_d ::= a\langle\overline{\rho}\rangle^{\downarrow} \qquad \xi_k ::= a\,[x]$$
$$\rho ::= a\langle\overline{\rho}\rangle \quad | \quad \rho \mid \rho \qquad \overline{\rho} ::= x \quad | \quad \rho \quad | \quad (a)\langle\overline{\rho}\rangle \quad | \quad \_$$

In this pattern language, the special pattern $\_$ matches anything.

For convenience, we write $a\langle x_1, \cdots, x_n\rangle$ for $a\langle \mathtt{1}\langle x_1\rangle \mid \cdots \mid \mathtt{n}\langle x_n\rangle\rangle$ where $\mathtt{1}, \cdots, \mathtt{n}$ only occur in these encodings.

We also write $a\langle\mathbf{0}\rangle$ for an argument a of a message in processes, and $a\langle\_\rangle$ in patterns. That is the process

$$(a\langle(b)\rangle \mid c\langle k\rangle \triangleright b\langle k\rangle) \mid a\langle d\rangle \mid c\langle k\rangle$$

corresponds to

$$(a\langle(b)\langle\_\rangle\rangle \mid c\langle k\langle\_\rangle\rangle \triangleright b\langle k\langle\mathbf{0}\rangle\rangle) \mid a\langle d\langle\mathbf{0}\rangle\rangle \mid c\langle k\langle\mathbf{0}\rangle\rangle$$

The matching functions are easily defined by induction:

$$\begin{aligned}
\mathtt{match}_m(a\langle\overline{\rho}\rangle, a\langle P\rangle) &\triangleq \mathtt{match}_r(\overline{\rho}, P)\\
\mathtt{match}_d(a\langle\overline{\rho}\rangle^{\downarrow}, a\langle P\rangle^{\downarrow_b}) &\triangleq \mathtt{match}_r(\overline{\rho}, P)\\
\mathtt{match}_u(a\langle\overline{\rho}\rangle^{\uparrow}, a\langle P\rangle^{\uparrow_b}) &\triangleq \mathtt{match}_r(\overline{\rho}, P)\\
\mathtt{match}_k(a\,[x]\,, a\,[P]) &\triangleq \{^P/_x\}\\
\mathtt{match}_r(\_, P) &\triangleq \{\}\\
\mathtt{match}_r(x, P) &\triangleq \{^P/_x\}\\
\mathtt{match}_r(\rho, P) &\triangleq \mathtt{match}(\rho, P)\\
\mathtt{match}_r((a)\langle\overline{\rho}\rangle, b\langle P\rangle) &\triangleq \{^b/_a\} \oplus \mathtt{match}_r(\overline{\rho}, P)
\end{aligned}$$

As an illlustration of the expressive power of the polyadic name-passing jK calculus, we consider an encoding of a π-calculus based instance (without hierarchical localities) of the generic membrane model developed by G. Boudol [4]. In this model, a locality has the form $a(S)[P]$, similar to that of M-calculus localities, with a control part S and a process part P. We thus consider an instance of the membrane model where both the control and process parts are written using the polyadic π-calculus. We refer the reader to G. Boudol's article in this volume for details about the membrane model.

The encoding of the π-calculus instance of the membrane model is given by the functions $[\![]\!]$, defined inductively in Figure 12 (where S, T are arbitrary control processes or plain processes, A, B are networks, i.e. parallel compositions of localities and network messages of the form $a\langle M\rangle$, and where M are local messages of the form $u\langle\widetilde{V}\rangle$, with u a channel name, V a value, which can be either a name or a process):

$$\begin{array}{ll}
[\![\mathtt{nil}]\!]_a \triangleq \mathbf{0} & [\![n]\!]_a \triangleq n \\
[\![u(\widetilde{x}).S]\!]_a \triangleq u\langle\widetilde{x}\rangle \triangleright [\![S]\!]_a & [\![!u(\widetilde{x}).S]\!]_a \triangleq u\langle\widetilde{x}\rangle \diamond [\![S]\!]_a \\
[\![u\langle\widetilde{V}\rangle]\!]_a \triangleq u\langle[\![\widetilde{V}]\!]_a\rangle & [\![\nu n.S]\!]_a \triangleq \nu n.[\![S]\!]_a \\
[\![\mathtt{out}\langle b, M\rangle.S]\!]_a \triangleq \mathtt{out}\langle b, [\![M]\!]_a\rangle.[\![S]\!]_a & [\![\mathtt{up}\langle M\rangle]\!]_a \triangleq \mathtt{up}\langle[\![M]\!]_a\rangle \\
[\![\mathtt{in}\langle P\rangle.S]\!]_a \triangleq \mathtt{in}\langle a, [\![P]\!]_a\rangle.[\![S]\!]_a & [\![S \mid T]\!]_a \triangleq [\![S]\!]_a \mid [\![T]\!]_a \\
[\![a(S)[P]]\!] \triangleq \nu c.a[\mathtt{MS}(a) \mid [\![S]\!]_a \mid c[\mathtt{MP}(a) \mid [\![P]\!]_a]] \mid \mathtt{Env} & \\
[\![A \parallel B]\!] \triangleq [\![A]\!] \mid [\![B]\!] & [\![a\langle M\rangle]\!] \triangleq \mathtt{in}\langle a, [\![M]\!]_a\rangle \\
[\![\nu n.A]\!] \triangleq \nu n.[\![A]\!] &
\end{array}$$

Fig. 12. Encoding the membrane calculus

The auxiliary processes $\mathtt{MS}(a)$, $\mathtt{MP}(a)$ and $\mathtt{Env}$ are defined as follows (note that they allow the incoming of messages from the outside—environment of the locality or the control process S—and from the inside—the plain process P):

$$\begin{aligned}
\mathtt{MS}(a) &\triangleq (\mathtt{in}\langle a, x\rangle^{\uparrow} \diamond x) \mid (\mathtt{up}\langle x\rangle^{\downarrow} \diamond S \mid x) \\
\mathtt{MP}(a) &\triangleq (\mathtt{in}\langle a, x\rangle^{\uparrow} \diamond x) \\
\mathtt{Env} &\triangleq \mathtt{out}\langle(b), x\rangle^{\downarrow} \diamond \mathtt{in}\langle b, x\rangle
\end{aligned}$$

As one can see, the notion of membrane readily translates into a pair of nested kells, with the outer one containing an encoding of the control part, and the inner one containing an encoding of the process part. It is important to note that the process $\mathtt{Env}$ above is coalescing, i.e. $\mathtt{Env} \mid \mathtt{Env} \sim^c \mathtt{Env}$, which ensures that the encoding is compositional.

5.2 FraKtal

We present in this section a calculus, called FraKtal, in which we can model several interesting features of a recent reflective component model called Fractal [5]. Fractal provides traditional notions of commponent-based software-engineering, namely components with input and output interfaces (or *ports*), which can be explicitly connected or disconnected during execution by means of bindings (or *connectors*). In addition, Fractal allows different forms of component introspection and intercession, such as adding and removing subcomponents, adding and removing interceptors on interfaces, controlling a component execution and life-cycle, etc. Interestingly, the reflective features in the Fractal model are introduced by means of a general component structure which distinguishes between the component membrane, which contains all the control functions, and the component content, which consists of other components (the subcomponents). This distinction between component membrane and component content is not dissimilar to that of the generic membrane model mentioned above.

The calculus we use to model components is a simple extension of the previous calculus with a construction that let us check that the argument of a message is *not* a given name, which we write $\overline{a}$. FraKtal also provide a way to bind such a name: $((m) \neq a)$ matches a name that is not a and binds it to m.

$$
\begin{array}{llll}
\xi ::= J \;\mid\; \xi_k \;\mid\; J \mid \xi_k & & J ::= \xi_m \;\mid\; \xi_d \;\mid\; \xi_u \;\mid\; J \mid J & \\
\xi_m ::= a\langle\overline{\rho}\rangle & \xi_u ::= a\langle\overline{\rho}\rangle^{\uparrow} & \xi_d ::= a\langle\overline{\rho}\rangle^{\downarrow} & \xi_k ::= a\,[x] \\
\rho ::= a\langle\overline{\rho}\rangle \;\mid\; \rho \mid \rho & \multicolumn{3}{l}{\overline{\rho} ::= x \;\mid\; \rho \;\mid\; (a)\langle\overline{\rho}\rangle \;\mid\; \overline{a}\langle\overline{\rho}\rangle \;\mid\; ((m) \neq a)\langle\overline{\rho}\rangle \;\mid\; \_}
\end{array}
$$

We similarly extend the matching functions, adding two cases for the helper function $\mathtt{match}_r$:

$$
\begin{aligned}
\mathtt{match}_r(\overline{a}\langle\overline{\rho}\rangle, b\langle P\rangle) &\triangleq \mathtt{match}_r(\overline{\rho}, P) \quad \text{if } a \neq b \\
\mathtt{match}_r(((m) \neq a)\langle\overline{\rho}\rangle, b\langle P\rangle) &\triangleq \{{}^{b}/_{m}\} \oplus \mathtt{match}_r(\overline{\rho}, P) \quad \text{if } a \neq b
\end{aligned}
$$

Sandboxes. We now present a way to isolate some computation from the environment as well as cleaning up the remains of this computation, in the form of a *sandbox*, defined as:

$$
\mathit{sandbox}\langle x, (\kappa)\rangle \triangleq \nu b.\nu r.\left(\begin{array}{l} \left(r\langle z\rangle^{\downarrow} \triangleright (b\,[y] \triangleright \mathbf{0}) \mid \kappa\langle z\rangle\right) \\ b\left[\left(\kappa\langle z\rangle^{\downarrow} \triangleright r\langle z\rangle\right) \mid b\,[x]\right] \end{array}\right)
$$

A sandbox works the following way: a firewall $\nu b.b\,[b\,[P]]$ is created, with an intermediate definition that waits for some message on κ supposed to contain the result of the computation. When such a message becomes available, it is consumed and forwarded to a private channel r (this channel being private, there is no risk that the computation in the sandbox, or some process in the environment, consumes it by mistake). This message triggers another rule that forwards it back to the κ channel outside of the sandbox as the final result, and creates a rule to garbage collect the sandbox.

Encoding Association Lists. Our association list encoding, defined in Figure 13, uses sandboxes for two reasons. First, it allows for handling lists atomically (for instance to check that a list does not contain some value before changing it). Second, it lets us clean up recursive definitions used to iterate on the list.

$$
\begin{aligned}
&\mathit{new\_list}\langle\kappa\rangle \diamond \nu\begin{pmatrix} l_{op}, g, s, r \\ l, \mathit{cons}, \mathit{nil}\end{pmatrix} . \left(\mathtt{List}\begin{pmatrix} l_{op}, g, s, r \\ l, \mathit{cons}, \mathit{nil}\end{pmatrix} \;\middle|\; \kappa\langle l_{op}, g, s, r\rangle\right) \\
\mathtt{List}\begin{pmatrix} l_{op}, g, s, r \\ l, \mathit{cons}, \mathit{nil}\end{pmatrix} &\triangleq \left(l_{op}\langle p, \kappa\rangle \diamond \mathtt{Lop}\begin{pmatrix} p, \kappa, g, s, r \\ l, \mathit{cons}, \mathit{nil}\end{pmatrix}\right) \\
\mathtt{Lop}\begin{pmatrix} p, \kappa, g, s, r \\ l, \mathit{cons}, \mathit{nil}\end{pmatrix} &\triangleq \nu\kappa' . \left(\begin{array}{c} l\langle x_l\rangle \rhd \mathit{sandbox}\left\langle \begin{pmatrix} p \mid l\langle x_l\rangle \mid \mathtt{L}\begin{pmatrix} g, s, r \\ l, \mathit{cons}, \mathit{nil}\end{pmatrix} \\ (\kappa\langle z\rangle \mid l\langle x_l'\rangle \rhd \kappa'\langle z, x_l'\rangle)\end{pmatrix}, \kappa' \right\rangle \\ (\kappa'\langle z, x_l'\rangle \rhd \kappa\langle z\rangle \mid l\langle x_l'\rangle)\end{array}\right) \\
\mathtt{L}\begin{pmatrix} g, s, r \\ l, \mathit{cons}, \mathit{nil}\end{pmatrix} &\triangleq \mathtt{Set}(s, l, \mathit{cons}) \mid \mathtt{Get}(g, l, \mathit{cons}, \mathit{nil}) \mid \mathtt{Rem}(r, l, \mathit{cons}, \mathit{nil}) \\
\mathtt{Set}(s, l, \mathit{cons}) &\triangleq s\langle (n), y, z\rangle \mid l\langle x_l\rangle \diamond l\langle \mathit{cons}\langle n\langle y\rangle, x_l\rangle\rangle \mid z \\
\mathtt{Get}(g, l, \mathit{cons}, \mathit{nil}) &\triangleq g\langle (n), (\kappa), x\rangle \mid l\langle x_l\rangle \diamond \mathtt{Get_i}\begin{pmatrix} n, \kappa, x, x_l \\ \mathit{cons}, \mathit{nil}\end{pmatrix} \mid l\langle x_l\rangle \\
\mathtt{Get_i}\begin{pmatrix} n, \kappa, x, x_l \\ \mathit{cons}, \mathit{nil}\end{pmatrix} &\triangleq \nu g_i . \left(\begin{array}{c|c} g_i\langle \mathit{cons}\langle n\langle y\rangle, z\rangle\rangle \diamond \kappa\langle y\rangle & \\ g_i\langle \mathit{cons}\langle \overline{n}\langle y\rangle, z\rangle\rangle \diamond g_i\langle z\rangle & g_i\langle x_l\rangle \\ g_i\langle \mathit{nil}\langle\rangle\rangle \diamond x & \end{array}\right) \\
\mathtt{Rem}(r, l, \mathit{cons}, \mathit{nil}) &\triangleq r\langle (n), (\kappa), x\rangle \diamond \mathtt{Rem_i}\begin{pmatrix} n, \kappa, x, x_l \\ l, \mathit{cons}, \mathit{nil}\end{pmatrix} \mid l\langle x_l\rangle \\
\mathtt{Rem_i}\begin{pmatrix} n, \kappa, x, x_l \\ l, \mathit{cons}, \mathit{nil}\end{pmatrix} &\triangleq \nu\begin{pmatrix} r_i \\ \mathit{rev}\end{pmatrix} . \\
&\qquad \left(\begin{array}{r} r_i\langle \mathit{cons}\langle n\langle y\rangle, z\rangle, z'\rangle \diamond \kappa\langle y\rangle \mid \mathit{rev}\langle z, z'\rangle \\ r_i\langle \mathit{cons}\langle ((m) \neq n)\langle y\rangle, z\rangle, z'\rangle \diamond r_i\langle z, \mathit{cons}\langle m\langle y\rangle, z'\rangle\rangle \\ r_i\langle \mathit{nil}\langle\rangle, z'\rangle \diamond x \mid \mathit{rev}\langle \mathit{nil}\langle\rangle, z'\rangle \\ \mathit{rev}\langle z, \mathit{cons}\langle x, z'\rangle\rangle \diamond \mathit{rev}\langle \mathit{cons}\langle x, z\rangle, z'\rangle \\ \mathit{rev}\langle z, \mathit{nil}\langle\rangle\rangle \diamond l\langle z\rangle \\ r_i\langle x_l\rangle \end{array}\right)
\end{aligned}
$$

Fig. 13. Operations on Lists

We can use such association lists to add a value to a key only if the key is not already present, as in (we assume that l_{op}, get, and set are known):

$$
\mathit{add}_1\langle (n), x, (\mathit{ok}), (\mathit{fail})\rangle \diamond \nu\kappa . \left(\begin{array}{l} l_{op}\left\langle \begin{pmatrix} \mathit{get}\langle n, \mathit{fail}, \mathit{set}\langle n, x, \kappa\langle \mathit{ok}\langle\rangle\rangle\rangle\rangle \\ (\mathit{fail}\langle y\rangle \rhd \kappa\langle \mathit{fail}\langle y\rangle\rangle)\end{pmatrix}, \kappa \right\rangle \\ (\kappa\langle z\rangle \rhd z)\end{array}\right)
$$

Encoding Components. The encoding we choose for components is very much inspired by the encoding of the membrane calculus above: the *controller* of a component is a kell that contains the *content* of the component, which is another kell. The controller may contain other subcomponents, which implement for instance *bindings* between interfaces. For instance, in the following configuration, a component A (whose membrane is named $\mathtt{A_m}$), exposes a client interface $\mathtt{Ac}$ that is bound to the server interface $\mathtt{Bs}$ of component $\mathtt{B}$. Following the Fractal approach, the binding is under control of $\mathtt{A}$.

$$
\begin{array}{l}
\mathtt{A_m}\left[\begin{array}{l|l}
\mathtt{bind}\left[\left(\mathtt{Ac}\langle x\rangle^{\uparrow} \diamond \mathtt{out_b}\langle \mathtt{Bs}\langle x\rangle\rangle\right)\right] & \\
\left(\mathtt{out_b}\langle y\rangle^{\downarrow} \diamond \mathtt{out}\langle y\rangle\right) & \mathtt{A}\left[\mathtt{Ac_i}\langle \mathtt{args}\rangle\right] \\
\left(\mathtt{Ac_i}\langle x\rangle^{\downarrow} \diamond \mathtt{Ac}\langle x\rangle\right) &
\end{array}\right] \\
\mathtt{B_m}\left[\begin{array}{l|l}
\left(\mathtt{Bs}\langle x\rangle^{\uparrow} \diamond \mathtt{Bs_i}\langle x\rangle\right) & \mathtt{B}\left[\cdots\right] \\
\mathtt{Bs_i}\langle \mathtt{args}\rangle &
\end{array}\right] \\
\left(\mathtt{out}\langle y\rangle^{\downarrow} \diamond y\right)
\end{array}
$$

After a number of reductions, the message $\mathtt{Ac_i}\langle \mathtt{args}\rangle$ on the internal interface bound to $\mathtt{Ac}$ reaches the internal interface bound to $\mathtt{Bs}$ and the final configuration is:

$$
\begin{array}{l}
\mathtt{A_m}\left[\begin{array}{l|l}
\mathtt{bind}\left[\left(\mathtt{Ac}\langle x\rangle^{\uparrow} \diamond \mathtt{out_b}\langle \mathtt{Bs}\langle x\rangle\rangle\right)\right] & \\
\left(\mathtt{out_b}\langle y\rangle^{\downarrow} \diamond \mathtt{out}\langle y\rangle\right) & \mathtt{A}\left[\right] \\
\left(\mathtt{Ac_i}\langle x\rangle^{\downarrow} \diamond \mathtt{Ac}\langle x\rangle\right) &
\end{array}\right] \\
\mathtt{B_m}\left[\begin{array}{l|l}
\left(\mathtt{Bs}\langle x\rangle^{\uparrow} \diamond \mathtt{Bs_i}\langle x\rangle\right) & \mathtt{B}\left[\cdots\right] \\
\mathtt{Bs_i}\langle \mathtt{args}\rangle &
\end{array}\right] \\
\left(\mathtt{out}\langle y\rangle^{\downarrow} \diamond y\right)
\end{array}
$$

We are currently investigating the modelling of control features of Fractal components, which require in particular the use of association lists presented above to store the list of interfaces (both client and server), the list of subcomponents, as well as the list of bindings between interfaces.

6 Conclusion and Related Work

We have introduced in this paper the Kell calculus, a family of higher-order process calculi with hierarchical localities, and studied two notions of (strong) equivalence for the kell calculus: a form of context bisimilarity and a notion of contextual equivalence, inspired, respectively, by Sangiorgi's contextual bisimilarity and barbed congruence for the higher-order π-calculus. We have given sufficient conditions on pattern languages to obtain a sound and complete co-inductive characterization of barbed congruence (or contextual equivalence). To the best of our knowledge this is the first time such a result is obtained for a higher-order calculus with hierarchical localities.

The Kell calculus is an attempt to simplify the M-calculus and to generalize it, through the introduction of a family of input pattern languages. Both the M-calculus and the Kell calculus highlight the importance of programmable membranes to deal with different forms of localities, and the need to enforce a principle of local actions in a calculus for mobile and distributed programming. These concerns are shared by the work on a generic membrane model, developed as part of the Mikado project [4]. We believe our work on the Kell calculus bisimulation semantics can be of direct use to develop the semantical theory of the Mikado generic membrane model.

A number of recent works on bisimulations for mobile agent systems focus on different variants of Mobile Ambients, and include e.g. work on a bisimulation-based equivalence for Mobile Ambients with passwords [26], work on a sound and complete co-inductive characterization of a contextual equivalence for New Boxed Ambients [7] and for the Calculus of Mobile Resources [17]. The work by Hennessy et al. on SafeDpi [18] contains a sound and complete characterization of (dependently) typed contextual equivalence for a higher-order process calculus with flat localities. All these works rely on a form of contextual bisimulation, but none of them support process passivation.

Apart from the M-calculus, which directly inspired the development of the Kell calculus, the only process calculus with localities which we know of that contains a feature related to process passivation is the Seal calculus. The migrate and replicate operator of the Seal calculus provides much of the expressive power of the Kell calculus passivation facility. The work closest to ours in terms of bisimulation semantics, is therefore the work by Castagna et al on congruences for the Seal calculus [10]. This work defines a notion of hoe-bisimilarity, which can be understood as an adaptation of Sangiorgi's higher-order context bisimilarity, and which is proved to be sound with respect to a natural contextual equivalence. Hoe-bisimilarity is not complete with respect to contextual equivalence in the Seal calculus, however.

Much work remains to be done on a bisimulation theory for the Kell calculus. In particular, we have recently obtained a tractable (i.e. finitary) co-inductive characterization of strong contextual equivalence for certain instances of the Kell calculus, as was done by Sangiorgi for the higher-order π-calculus [31] and recently extended by Jeffrey and Rathke in [20]. We plan to extend this result to the weak and typed cases.

We believe the connection between the Kell calculus and component-based programming, which was just sketched in the last section of this paper, to be very promising. We are currently working on a full Fractal interpretation in FraKtal, which should allow us to leverage recent results in type systems for locality-based process calculi for component-based programming.

Acknowledgments. This paper has benefited from several discussions on the Kell calculus, over the past year, with David Teller, Daniel Hirschkoff, Tom Hirschowitz, Matthew Hennessy, Vasco Vasconcelos, Eugenio Moggi, Gérard Boudol, and Xudong Guan, which have helped clarify many points, and correct several errors. The contribution of all these individuals is greatly appreciated. The research reported in this paper has been supported in part by the IST Mikado project (IST-2001-32222).

References

1. J. Aldrich, C. Chambers, and D. Notkin. Architectural Reasoning in ArchJava. In *Proceedings 16th European Conference on Object-Oriented Programming (ECOOP)*, 2002.

2. R. Amadio. An asynchronous model of locality, failure, and process mobility. Technical report, INRIA RR-3109, INRIA Sophia-Antipolis, France, 1997.
3. F. Barbanera, M. Bugliesi, M. Dezani-Ciancaglini, and V. Sassone. A calculus of bounded capacities. In *ASIAN 03*, volume 2896 of *LNCS*, pages 205–223. Springer, 2003.
4. G. Boudol. A Generic Membrane Model. *this volume*, 2004.
5. E. Bruneton, V. Quéma T. Coupaye, M. Leclercq, and J.B. Stefani. An Open Component Model and its Support in Java. In *Proceedings 7th International Symposium on Component-based Software Engineering (CBSE 2004), LNCS 3054*. Springer, 2004.
6. M. Bugliesi, G. Castagna, and S. Crafa. Boxed ambients. In *4th International Symposium on Theoretical Aspects of Computer Software (TACS)*, 2001.
7. M. Bugliesi, S. Crafa, M. Merro, and V. Sassone. Communication Interference in Mobile Boxed Ambients. In *Proceedings of the 22nd Conference on Foundations of Software Technology and Theoretical Computer Science (FST-TCS '02*, volume LNCS 2556. Springer, 2002.
8. M. Carbone and S. Maffeis. On the Expressive Power of Polyadic Synchronization in π-calculus. *Electronic Notes in Theorectical Computer Science, vol. 68, no 2*, 2002.
9. L. Cardelli and A. Gordon. Mobile Ambients. *Theoretical Computer Science, vol. 240, no 1*, 2000.
10. G. Castagna and F. Zappa. The Seal Calculus Revisited. In *Proceedings 22th Conference on the Foundations of Software Technology and Theoretical Computer Science*, number 2556 in LNCS. Springer, 2002.
11. I. Castellani. Process algebras with localities. In *Handbook of Process Algebra, J. Bergstra, A. Ponse and S. Smolka (eds)*. Elsevier, 2001.
12. D. Clarke and T. Wrigstad. External Uniqueness is Unique Enough. In *Proceedings 17th European Conference on Object-Oriented Programming (ECOOP)*, 2003.
13. M. Coppo, M. Dezani-Ciancaglini, E. Giovannetti, and I. Salvo. $\mathbf{M}^3$: Mobility types for mobile processes in mobile ambients. In *CATS 2003)*, volume 78 of *ENTCS*, 2003.
14. S. Dal-Zilio. Mobile Processes: A Commented Bibliography. In *Modeling and Verification of Parallel Processes, 4th Summer School, MOVEP 2000*, volume 2067 of *LNCS*. Springer, 2001.
15. C. Fournet. *The Join-Calculus*. PhD thesis, Ecole Polytechnique, Palaiseau, France, 1998.
16. C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, and D. Remy. A calculus of mobile agents. In *In Proceedings 7th International Conference on Concurrency Theory (CONCUR '96), Lecture Notes in Computer Science 1119*. Springer Verlag, 1996.
17. J.C. Godskesen, T. Hildebrandt, and V. Sassone. A calculus of mobile resources. In *Proceedings 13th International Conference on Concurrency Theory (CONCUR 02)*, 2002.
18. M. Hennessy, J. Rathke, and N. Yoshida. Safedpi: a language for controlling mobile code. Technical Report 2003:02, University of Sussex, 2003. Extended abstract presented at Foundations of Software Science and Computation Structures - 7th International Conference, FOSSACS 2004.
19. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. Technical report, Technical Report 2/98 – School of Cognitive and Computer Sciences, University of Sussex, UK, 1998.
20. A. Jeffrey and J. Rathke. Contextual equivalence for higher-order π-calculus revisited. In *Proceedings, 19th Conference on the Mathematical Foundations of Programming Semantics*, 2003.
21. G. Leavens and M. Sitaraman (eds). *Foundations of Component-Based Systems*. Cambridge University Press, 2000.
22. L. Leth and B. Thomsen. Some facile chemistry. *Formal Aspects of Computing Vol.7, No 3*, 1995.

23. F. Levi and D. Sangiorgi. Controlling interference in ambients. In *Proceedings 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2000)*, 2000.
24. N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
25. N. Medvidovic and R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering, vol. 26, no. 1*, 2000.
26. M. Merro and M. Hennessy. Bisimulation congruences in safe ambients. In *29th ACM Symposium on Principles of Programming Languages (POPL), Portland, Oregon, 16-18 January*, 2002.
27. R. Milner. *Communicating and mobile systems : the π-calculus*. Cambridge University Press, 1999.
28. R. De Nicola, G.L. Ferrari, and R. Pugliese. Klaim: a Kernel Language for Agents Interaction and Mobility. *IEEE Trans. on Software Engineering, Vol. 24, no 5*, 1998.
29. R. De Nicola, G.L. Ferrari, R. Pugliese, and B. Venneri. Types for Access Control. *Theoretical Computer Science, Vol. 240, no 1*, 2000.
30. A. Ravara, A. Matos, V. Vasconcelos, and L. Lopes. Lexically scoping distribution: what you see is what you get. In *FGC: Foundations of Global Computing*, volume 85(1) of *ENTCS*. Elsevier, 2003.
31. D. Sangiorgi. Bisimulation for higher-order process calculi. *Information and Computation,Vol. 131, No 2*, 1996.
32. D. Sangiorgi and D. Walker. *The π-calculus: A Theory of Mobile Processes*. Cambridge University Press, 2001.
33. A. Schmitt and J.B. Stefani. The M-calculus: A Higher-Order Distributed Process Calculus. In *Proceedings 30th Annual ACM Symposium on Principles of Programming Languages (POPL)*, 2003.
34. P. Sewell and J. Vitek. Secure Composition of Insecure Components. *Journal of Computer Security*, 2000. Invited submission for a CSFW00 special issue.
35. J.B. Stefani. A calculus of kells. In Vladimiro Sassone, editor, *Proceedings International Workshop on Foundations of Global Computing, Electronic Notes in Theoretical Computer Science*, volume 85(1). Elsevier, 2003.
36. B. Thomsen. A Theory of Higher Order Communicating Systems. *Information and Computation, Vol. 116, No 1*, 1995.
37. Vasco T. Vasconcelos. A note on a typing system for the higher-order π-calculus. September 1993.
38. P. Wojciechowski and P. Sewell. Nomadic Pict: Language and Infrastructure. *IEEE Concurrency, vol. 8, no 2*, 2000.
39. N. Yoshida and M. Hennessy. Assigning types to processes. In *15th Annual IEEE Symposium on Logic in Computer Science (LICS)*, 2000.

A Software Framework for Rapid Prototyping of Run-Time Systems for Mobile Calculi*

Lorenzo Bettini[1], Rocco De Nicola[1], Daniele Falassi[1], Marc Lacoste[2], Luís Lopes[3], Licínio Oliveira[3], Hervé Paulino[4], and Vasco T. Vasconcelos[5]

[1] Dipartimento di Sistemi e Informatica, Università di Firenze
[2] Distributed Systems Architecture Department, France Telecom R & D
[3] Departamento de Ciência de Computadores, Faculdade de Ciências, Universidade do Porto
[4] Departamento de Informática, Faculdade de Ciências e Tecnologia, Univ. Nova de Lisboa
[5] Departamento de Informática, Faculdade de Ciências, Universidade de Lisboa

Abstract. We describe the architecture and the implementation of the MIKADO software framework, that we call IMC (*Implementing Mobile Calculi*). The framework aims at providing the programmer with primitives to design and implement run-time systems for distributed process calculi. The paper describes the four main components of abstract machines for mobile calculi (node topology, naming and binding, communication protocols and mobility) that have been implemented as Java packages. The paper also contains the description of a prototype implementation of a run-time system for the Distributed Pi-Calculus relying on the presented framework.

1 Introduction

It has been widely argued that mobility will be an important technology for applications over a global network. The main breakthrough is that global applications may exchange *mobile code* [9, 33], not just data. A particular instance of mobile code is the concept of *mobile agents* [14, 19, 36]: software units that can suspend their execution and migrate to new hosts, where they can resume their execution.

The programming paradigm based on mobile agents is different from remote evaluation or code on-demand in that the code does not need to be sent and retrieved explicitly: the agent migrates autonomously somewhere else and continues executing there. An agent is self contained in that it possesses all the data it needs to execute and migrate, since this information is typically carried with the agent during migration. Decisions of executing and moving are taken according to information supplied by the programmer of the agent. Agents may also autonomously decide to take different choices due to contextual events such as, temporary unavailability of networks or not responding hosts.

Dealing with mobile code and mobile agents raises a number of issues such as: packing of moving agents, security, protocols, naming, network architecture. We are

* This work has been funded by EU-FET on Global Computing, project MIKADO IST-2001-32222. The funding body is not responsible for any use that might be made of the results presented here.

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 179–207, 2004.

developing a generic framework called IMC (*Implementing Mobile Calculi*) that can be used as a kind of middleware for the implementation of different mobile programming systems. Such a framework aims at being as general as possible. It can be, and indeed has been, used to implement existing systems (KLAVA [5], Safe Ambients [32], JCL [12] and DITYCO [26]) on top of it. But it also provides the necessary tools for implementing new languages directly derived from calculi for mobility.

The implementer of a new language would need concentrating on the parts that are really specific of his system, while relying on the framework for the recurrent standard mechanisms. The development of prototype implementations should then be quicker and the programmers should be relieved from dealing with low level details.

For the sake of dissemination and portability, the framework is being developed as Java packages. Thus, the used virtual machine technology is the one based on the Java Virtual Machine. This choice is also motivated by the fact that most existing mobile code systems are based on Java.

The proposed middleware framework aims at implementing (or, at least, specifying) all the required functionalities for arbitrary components to communicate and move in a distributed setting. Four abstractions have been isolated as being fundamental to this goal and each of them has been implemented as a sub-package of our IMC Java package:

- node topology
- naming and binding
- communication protocols
- mobility

The purpose of the sub-package for **Node Topology** is to describe the encoding of the topological structure of the network and to take into account the effect of distributed computations performing changes in its overall structure. Its main components deal with primitives for connection and disconnection, node creation and deletion, domain specific node coordination (membrane, guardian, etc.), node-based decentralized topology and actual implementation of nodes and node naming.

The purpose of the sub-package for **Naming and Binding** is to define a uniform way to designate and interconnect the set of objects involved in the communication paths between computational nodes. We call such a set of objects a *binding*. Its main components are designed to deal with primitives for name creation and deletion, typing and compatibility checking, policies for name resolution (static, dynamic, mixed, bindings, ...) and name marshalling and un-marshalling.

The purpose of the sub-package for **Communication Protocols** is to identify both the abstractions and the primitives for logical and physical node connectivity, as well as the strategies that can be used to capture and perform communications between computational nodes. Its components are designed to deal with abstract, possibly secure, send and receive primitives, marshalling of messages at network level, session management, connection checking and multicasting.

The purpose of the sub-package for **Code Mobility** is to provide the basic functionalities for making code mobility transparent to the programmer; all issues related to code marshaling and code dispatch are handled automatically by the classes of the framework. Its components are designed to deal with object marshalling, code migration, and dynamic loading of code.

The general aim of the framework is to assist both the designer and the programmer of a virtual machine or run-time system implementing a domain-based programming model. The four components of the framework are connected and cooperate in order to implement the abstract representation of a distributed application with mobile code. Thus, for instance, the `topology` package relies on the `protocols` package to actually communicate over the network and the `protocols` package, in turn, relies on the `mobility` package to create packets containing migrating code, and so on. We observe that these cooperations are made through interfaces and abstract classes. Nonetheless, IMC already provides concrete implementations for the standard and most used functionalities that should fit most Java mobile framework requirements (e.g., Java byte-code mobility and TCP/IP sockets).

The user of the IMC package can then customize parts of the framework by providing its own implementations for the interfaces used in the package. In this respect, the IMC framework will be straightforward to use if there is no need of specific advanced features. Nevertheless, the frameworks is open to customizations if these are required by the specific mobility system one is willing to implement. For example, the TyCO system makes use of its own code dispatch strategy and overrides the standard Java byte-code mobility. Customization of the framework can be achieved seamlessly thanks to design patterns such as *factory method* and *abstract factory* [13] that are widely used throughout the package.

The above mentioned sub-packages have been developed over the mikado sites by taking advantage of CVS server organized as a single project `org.mikado.imc` structured with four tasks:

- `org.mikado.imc.topology`
- `org.mikado.imc.naming`
- `org.mikado.imc.protocols`
- `org.mikado.imc.mobility`

The rest of the paper contains the detailed description of the four sub-packages and ends with an experimental implementation of $D\pi$, the only model considered within the Mikado project that has not yet been implemented.

2 Node Topology

The purpose of this part of the framework is to describe the encoding of the topological structure of the network and to take into account the effect of distributed computations performing changes in its overall structure.

Motivation

The notion of node appears in most existing implementations of mobile calculi, such as, e.g., [4, 5, 12, 32, 34]. However, the internal structure of the node itself is programming-model specific. The computational nodes can include data structures ranging from definitions (TyCO, Jcl/JoCaml, CLAM), processes (X-Klaim/Klava, TyCO, SAM, Jcl/JoCaml, CLAM) channels (TyCO, SAM, Jcl/JoCaml, CLAM), objects (TyCO) and tuple spaces (X-Klaim/Klava).

From this observation we designed the Node Topology package so that it would provide both a programming abstraction and a generic concrete implementation for a node whilst not providing any details of the specific implementation of the virtual machine that will run on it.

Another common property of the current mobile calculi implementations is that the nodes are designated to use some form of unique identifier. It is noteworthy that the structure of the node identifier is itself implementation specific. Also, node identifiers must be created dynamically when new nodes are added to a network.

Thus, we require the Node Topology framework to provide both an abstraction for a node identifier that encapsulates its implementation details and, some mechanism to create fresh node identifiers in accordance with some implementation-specific format.

In the existing implementations of mobile calculi, the topological organization of nodes is either hierarchical, i.e., tree-structured, or flat. For example, SAM, Jcl/JoCaml, X-Klaim/Klava and CLAM use a tree-structured topology of nodes, while TyCO nodes are organized according to a flat structure. Primitives for expressing a topological hierarchy of nodes can easily be used to reflect a flat organization of nodes by adding a root (virtual) location whose children are the given nodes. Thus, we feel that the framework should support hierarchical node composition patterns. This in turn implies providing tools to navigate through the network hierarchy and retrieve information about its structure.

Finally, some process calculi have reduction rules that imply adding new nodes to the hierarchy or removing existing nodes (or moving them). This can be due to the strict implementation of the reduction rules, or to new components dynamically being introduced into a running system. Thus, the Node Topology framework should provide primitives to add new nodes to the network hierarchy or to remove existing nodes.

Design

The above requirements lead to the following design for the Node Topology sub-package in the form of Java interfaces. Concrete and generic default implementations of these interfaces are also provided in the sub-package and named by prefixing the interface name with `Mikado` (e.g., a default implementation for interface `Node` is provided by the class `MikadoNode`).

```
interface NodeIdentifier {
  public Object getIdentifier();
}

interface Node {
  public String getNodeName();
  public NodeIdentifier getNodeIdentifier();
  public Object getImplementation();
  public NodeIdentifier getParentNode();
  public NodeIdentifier [] getChildNodes();
  public void connect(NodeIdentifier nodeId);
  public void disconnect();
```

```
   public Registry getRegistry();
}
```

The `NodeIdentifier` interface defines a generic way to identify a computational node component. Implementation-specific node identifiers may be obtained by designing specific classes implementing that interface. The `Node` interface describes the basic computational nodes which may also be called location, site or agent, according to the underlying programming model. It contains a reference to its node identity interface, as well as a reference to an object holding the internal implementation of the node. That particular structure is implementation specific, and is not part of the Node Topology sub-package.

The requirements of a hierarchical network topology and support for editing such a hierarchy lead to the introduction of topology management functionality directly within the `Node` interface. Thus, the methods `getParentNode()` and `getChildNodes()` allow a node to inspect its network neighborhood. The methods `connect()` and `disconnect()` handle a node's connection to a specific point (as a sub-node) of the hierarchy and its disconnection when leaving the network or migrating to another point in the hierarchy.

The method `getRegistry()` provides access to a node's table of exported resources (e.g., channels) and will be further commented when describing the Naming and Binding sub-package.

A special node in a Mikado Network, the `NetworkServer`, acts as a portal where all nodes adhering to a computation must first register. The `NetworkServer` handles the mappings between nodes and their physical locations (e.g., IP addresses) in a Mikado Network. The network server accepts different implementations and network configurations that can be specified at startup by providing an implementation for the interface `NetworkServerImpl` and a class holding the network configuration called `Preferences`. These settings and the current server handle can be obtained through a set of methods (`getImpl()`, `getServer()` and `getConfig()`).

```
class NetworkServer {
   NetworkServer(String [] args);
   static NetworkServer getServer();
   static NetworkServerImpl getImpl();
   Preferences getConfig();
   ...
}
```

Examples

IMC's `topology` revolves around the interface `Node`. This interface represents a running instance of a virtual machine and includes a set of operations that manipulate that same VM instance. To enforce interoperability between the different virtual machines that may end up subclassing IMC, a base implementation for the interface `Node` has already been provided, in the form of the `MikadoNode` class.

In TyCO, the abstraction for a node in a network is implemented by a class `Site`. Since `Site` cannot subclass any other class (it already subclasses TyCO's `TyCOVirtualMachine` class and Java does not allow multiple inheritance), we create a wrapper class, TyCONode, that holds an instance of the TyCOVirtualMachine class.

```
public class TyCONode extends MikadoNode implements ... {
  Object virtualMachine;
  TyCONode (String name, Object virtualMachine) {
    super(name);
    this.virtualMachine = virtualMachine;
  }
  public Object getImplementation() {
    return this.virtualMachine;
  }
  ...
}
```

To create a new node running a TyCO virtual machine and add it to a network of running nodes one has to create a new instance of the class TyCONode supplying the node's name and the virtual machine running in it. To attach/detach the node to/from the network we need only to call the superclass' (MikadoNode) methods: connect() and disconnect().

```
public class NodeManager {
  ...
  void newNode(String name, Assemble assembly) {
    TyCOVirtualMachine virtualMachine = new TyCOVirtualMachine(name, assembly);
    TyCONode node = new TyCONode(name, virtualMachine);
    node.run();
    node.connect();
  }
  ...
}
```

We assume the existence of a connect() wrapper in the TyCONode that skips the NodeIdentifier argument and automatically connects the new TyCO node to the NetworkServer that serves as the root of the flat network topology of the TyCO network.

An additional, optional, step is the inclusion of a TopologySecurityManager, which controls access to MikadoNode's functionality. It allows an operation to be blocked or allowed, based on any desired security policy. The default TopologySecurityManager for TyCO would enforce rules such as: a node can only connect to the NetworkServer (its parent) and that it cannot accept connections (since the topology is flat).

The IMC infrastructure already contains a NetworkServer that handles the mappings between the node names and their physical location in a network. This enables a straightforward implementation of the TyCO's Name Server by using it as the virtual root of the TyCO flat network topology and extending the class with functionality for registering and type-checking top level exported channels.

3 Naming and Binding

The purpose of this part of the framework is to define a uniform way to designate and interconnect the set of objects involved in the communication paths between computational nodes. We shall call such a set of objects a *binding*.

Motivation

The fundamental concept to be provided by this sub-package is that of a *referenceable* object. Such an object is an abstraction for the fundamental communication peers in process calculi such as channels or definitions. A referenceable object is always associated with a unique network-wide *identifier*. Each resource identifier is uniquely associated with a *naming context* in a network.

A feature common to current mobile calculi implementations is the use of the export/bind programming pattern to make objects available in a network and to get an access path for such objects. This pattern is so pervasive that the Naming and Binding sub-package provides a *registry* abstraction that, for a given managed name, should be able to make it available to the network by registering it and to create an access path towards *the* object designated by that name. Thus, the registry is responsible for keeping the mappings between identifiers and referenceables for a given node in a network.

The above considerations offer a generic and uniform view of bindings, clearly separating object identification from object access.

Design

We now describe a minimal set of interfaces for dealing with naming and binding based on the above requirements:

```
 public interface UID {
 public NamingContext getContext();
 public String getName();
 public String getEncoded();
 public NodeIdentifier getNodeIdentifier();
 public String toString();
}

public interface NamingContext {
 public String getName();
}
```

The `UID` interface represents the generic notion of a network wide unique identifier used to designate some object relatively to a given naming context, such as a channel in process calculi. Identifiers are implementation dependent. The interface contains a reference to its naming context. The `Naming Context` interface represents a set of `Referenceable` (see below) objects in a Mikado Network that is identified by a string.

```
public interface Referenceable {
  public NamingContext getContext();
  public UID getUID();
  public void handleData(ProxyRequest request);
  public void marshall();
  public void unmarshall();
}
```

The `Referenceable` interface must be implemented by any object in a Mikado Network that is to be exported and bound during a computation. The `handleData()` method is used to receive requests from object proxies (`Proxy`) elsewhere in the network. The `marshall()` and `unmarshall()` methods can be used, respectively, to prepare a reference for network dispatch and to restore a reference after traveling through the network. These methods typically call a `Marshaller` implementation from the `protocols` sub-package to perform some level of packing/unpacking (see Section 4).

```
public abstract class Proxy {
  private UID uid;
  protected Proxy(UID uid);
  public UID getUID();
  public abstract void dispatch(Serializable request);
  public abstract void marshall();
  public abstract void unmarshall();
}
```

```
public interface ProxyRequest {
  public abstract NodeIdentifier getPeer();
  public abstract UID getUID();
  public abstract Serializable getRequest();
}
```

The `Proxy` interface describes the functionality associated with a proxy for a referenceable object. The method `dispatch()` handles communication by redirecting it to the corresponding referenceable object. Methods `marshall()` and `unmarshall()` have similar functions to the referenceable side. The interface `ProxyRequest` allows a referenceable object to obtain basic topological and naming information about a proxy sending data from another node and the data itself.

```
public interface Registry {
  public void export(Referenceable ref);
  public void unexport(Referenceable ref);
  public Proxy bind(NodeIdentifier id, String name, NamingContext context);
  public Referenceable getRef(UID uid);
  public Referenceable [] getAllRefs();
}
```

Object access is provided for by the interface `Registry` which must be implemented by any class that exports or binds objects in a Mikado Network. The interface includes the `export()` method to create a new name in a given context and make it bindable in a network. The `unexport()` method cancels an `export` operation by making an identifier no longer valid within a naming context. In other words, the mapping identifier-referenceable object designated by that identifier is broken. The `bind()` method returns a local Proxy associated with a given Referenceable object at a node `id`, with a given `name` and naming context `context`. This Proxy allows direct communication with the proxy for the Referenceable object in the Mikado Network.

All the required communication between referenceable objects and their proxies is provided via the protocols sub-package of the framework.

Examples

A default implementation of some of the `Naming` and `Binding` package interfaces (`Proxy`, `ProxyRequest`, `NamingContext` and `Registry`) is already provided in IMC (`MikadoProxy`, `MikadoProxyRequest`, `MikadoNamingContext` and `MikadoRegistry`, respectively).

The fundamental step in implementing TyCO on the IMC framework is the realization that TyCO's referenceable objects are instances of the class `Channel`. Also, given the flat topology of TyCO networks, the implementation requires only a single naming context identified by the string `"tyco"`.

In this approach, we make each exported TyCO channel in a running virtual machine implement the Referenceable interface and allow other nodes in the network to communicate with it directly by using proxies through the `Proxy` interface. The most important method in this implementation is `handleData()`. This method handles proxy requests to the channel from proxies elsewhere in the TyCO network. The incoming requests, messages or objects are either enqueued in the channel queue or reduced immediately if an adequate message-object redex forms.

```
public class Channel extends ... implements Referenceable {
  ...
  public void handleData(ProxyRequest request) {
    // unpack the request and check whether it is an object or a message
    Frame frame = unpack(request.getRequest());
    // run code according to case
    if( status == 0 ) {   // channel is empty
      enqueue(frame);
      if ( frame.isObject() )
        status++;
      else
        status−−;
    } else if ( status < 0 ) { //channel has messages
      if ( frame.isObject() ) {
        Frame message = dequeue();
        reduce(frame, message);
        status++;
      } else {
        enqueue(frame);
        status−−;
      }
    } else if ( status > 0 ) { // channel has objects
      if ( frame.isObject() ) {
        enqueue(frame);
        status++;
      } else {
        Frame object = dequeue();
        reduce(object, frame);
        status−−;
      }
    }
  }
  ...
}
```

TyCO represents channels in a distributed computation in two distinct formats reflecting their current position relative to their lexical bindings. A local channel to a node is represented as a JVM heap reference. A remote channel is represented in a network format containing information about the node (its name) it originated from and its local reference there. When, say, a message is sent to a channel in a program running at some node of a TyCO network, the internal representation of the channel is first checked to see if the channel is local to the node or if it is a remote channel. If the channel is remote, a `bind` operation is required to get a proxy to handle remote interaction. Thus, a TyCO Virtual machine instruction to handle message delivery would look like this:

```
void sendMessage(Channel channel, Label label, Value[] args) {
  if (channel.isLocal()) {
    // code for local handling, similar to above code for handleData()
    ...
  } else {
    // get a proxy object for communication with the real channel
    UID uid = channel.getUID();
    NodeIdentifier nodeId = uid.getNodeIdentifier();
    String name = uid.getName();
    NamingContext context = uid.getContext();
    Proxy proxy = getMikadoNode().getRegistry().bind(nodeId, name, context);
    // pack message in a Serializable object and send it to the channel
    SerializablePacket packet = new SerializablePacket(channel, label, args);
    proxy.dispatch(packet);
  }
}
```

Exporting and importing top-level channels in TyCO involves two operations in the TyCO Virtual Machine that interacts with the NetworkServer. When we `export` one channel from a node we make its access information available to other nodes in the network. Such an operation might be implemented using the above abstractions as:

```
void export(Channel channel) {
  getMikadoNode().getRegistry().export(channel);
}
```

The complementary operation in which we `import` a top-level channel from some node in a TyCO network requires the name of the channel being requested and the node it resides in. The operation simply requests a proxy for the remote channel based on the name of the channel and of the node:

```
Proxy import(String node, String name) {
  NodeIdentifier nodeId = getMikadoNode().resolve(node);
  NamingContext context = new TycoNamingContext();
  return getMikadoNode().getRegistry().bind(nodeId, name, context);
}
```

4 Communication Protocols

This part of the IMC framework intends to identify the primitives and the communication strategies for logical and physical node connectivity. The general aim is to assist the architect of a run-time system for a distributed process calculus in the implementation of new communication protocols between computational nodes.

Motivation

Existing implementations [3] of some common distributed process calculi [6] are characterized by a flurry of communication protocols and of programming languages. In first approximation, the protocols can be split into two families: high-level protocols such as Java RMI are well integrated with the Java Virtual Machine environment and take advantage of the architectural independence provided by Java (SAM [32] implementation of Safe Ambients [25]); protocols closer to hardware resources such as TCP/IP are accessible, either directly in Java (X-Klaim/Klava [5]) or in other programming languages allowing easier manipulation of system resources such as OCaml (Jcl [12] and JoCaml [24]) or C (DiTyCO [26]). Marshalling strategies range from dedicated byte-code structures (Jcl, JoCaml, DiTyCO) to Java serialization (SAM, X-Klaim/Klava).

Thus, a generic communication framework to build prototype implementations of process calculi cannot restrain itself to a fixed set of interaction primitives or marshalling strategies. Instead, a middleware like IMC should be flexible enough to support multiple marshalling strategies and communication protocols. The framework should also aim at minimality to introduce new communication protocol support with little effort, in any case without need to re-implement a new communication library: either by realizing specialized implementations of the framework interfaces, or by defining framework increments which will complement the IMC core interfaces and libraries.

A number of minimal platforms for flexible communications have already been implemented [10, 11, 15, 18, 20, 28] where objects interact transparently through remote method invocations on well-defined interfaces. Their originality compared to CORBA-like or Java RMI-based infrastructures is to provide a core framework for building different types of middleware using the notion of flexible *bindings*. Creating a new binding should be understood as setting up access and communication paths between components of a distributed system with a wide variety of semantics: mobile, persistent, with QoS guarantees, etc. An adaptable communication framework should provide primitives to define bindings with various semantics, and to combine them in flexible ways. With simple architectural principles such as separating marshalling from protocol implementation, or threading from resource management, those middleware have shown how to dynamically introduce new protocols or control the level of resource multiplexing. In the IMC communication framework design, an important decision was to leverage the previously described naming and binding framework for network protocols in order to achieve adaptable forms of communication transparency needed when implementing a specific process calculus. The communication framework enables the definition of customized protocol stacks by a flexible composition of micro-protocols. In practice, the implementation of a new process calculus will most likely use TCP/IP as lowest layer of interface to the network. Thus, the IMC communication framework provides support for TCP/IP bindings, but can be easily extended to other protocols.

The IMC communication framework is composed of two main sub-packages:

- Sub-package `org.mikado.imc.protocols.apis` contains the interfaces describing the main abstractions for communication, e.g. sessions, protocols, marshallers, ...
- Sub-package `org.mikado.imc.protocols.libs` contains the classes which implement those interfaces and offer support for flexible TCP/IP bindings.

In what follows, we describe the main abstractions and interfaces of the IMC communication framework. We then illustrate over a simple example – a small client/server authentication protocol called "knock-knock" – how protocol and session objects can be combined to implement new communication protocols taking advantage if IMC.

Design

The communication framework builds upon the IMC abstractions for naming and binding such as identifiers, references, and naming contexts. Protocol-specific abstractions are inspired from the x-kernel [17] and JONATHAN [20] communication frameworks and are represented by the interfaces below:

```
public interface Protocol {}

public interface ProtocolGraph {
  public SessionIdentifier export(Session_Low session);
}

public interface SessionIdentifier {
  public Protocol getProtocol();
  public Session_High bind(Session_Low session);
}

public interface Session_High {
  public void send(Marshaller message);
}

public interface Session_Low {
  public void send(UnMarshaller message, Session_High session);
}
```

A *protocol* represents network protocols like TCP, IP, or GIOP, and provides a naming context for a particular kind of interfaces called *sessions*. It manages names called *session identifiers* to designate those interfaces.

The structure of a protocol stack is captured by a *protocol graph*. This directed acyclic graph composed of protocol nodes describes the path to be followed by messages when they are sent over the network, or received. A given session can be exported to inform the communication layers that it is willing to accept messages: a call on the `export` method at the root of a protocol graph will issue recursively the appropriate calls on each node of its sub-graphs. A session identifier is then returned to designate the exported session. To communicate with the exported session, a client just needs to call the `bind` method on the returned session identifier, which will provide a surrogate the client can use to send messages to the exported server session.

A *session* is an abstract representation of a communication channel. A session object is dynamically created by a protocol and lets messages be sent and received through the communication channel it stands for using that protocol. It has higher and lower interfaces to send messages down and up a protocol stack which may be viewed as a stack of sessions.

Exported session objects are designated using *session identifiers*. Their internal structure is protocol-specific. For instance, a TCP/IP session identifier encapsulates a host name and port number. Session identifiers are created when exporting a server-side session and then transmitted over the network. On the client side, they allow to establish communication channels by invoking the `bind` operation, with an optional session parameter to receive messages sent by the remote server-side session.

Sessions and Connections. Messages can navigate through a protocol stack using the session interfaces (`Session_High` and `Session_Low`) with a single method to perform the message sending operation. A `Session_High` object is used to send messages down to the network. It will usually be a surrogate for a `Session_Low` type of session, which has been exported to a `Protocol` instance and is designated by a `SessionIdentifier` interface. A `Session_High` instance may be obtained by invoking the `bind` operation on a session identifier representing a `Session_Low` interface: it is thus a surrogate, or a proxy, for that interface. A `Session_Low` object is used to forward messages coming from the network to their actual recipient. `Session_Low` is also the type of interfaces exported to protocols, and designated by session identifiers. The additional parameter in the `send` method represents the sender, and may be used to send a reply, if necessary.

Each session contains a lower-level abstraction of a communication channel called a *connection*. It typically encapsulates a regular socket, and provides operations to read and write to the socket. Client-side or server-side connections may be built on demand using *connection factories*, for instance on an incoming connection request from a client. A *connection manager* keeps track of idle and active connections, and delegates the creation of new connections to a connection factory.

To facilitate concurrent programming within sessions, the framework also offers basic primitives for activity management and their scheduling according to various criteria such as priorities, deadlines, etc.

Marshalling. Marshallers and unmarshallers are used as high-level and encoding-independent representations of messages that are about to be sent or received. The `Marshaller` interface is described below:

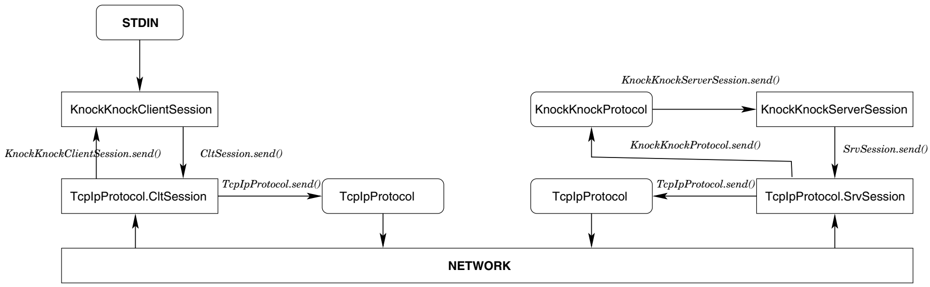


Fig. 1. Session and Protocol Objects in the “Knock-Knock” Protocol

```
  public void writeBoolean(boolean b);
  public void writeChar(char c);
  ...
  public void writeReference(Object obj);
  public void writeCode(MigratingCode code);
  public boolean isLittleEndian();
  public void close();
}
```

The `UnMarshaller` interface is similar but with read instead of write operations. The communication framework allows to customize the marshalling and unmarshalling of messages, by including interfaces for the management of *chunks*, or fragments of byte arrays which are chained together to form messages. The use of chunks helps avoiding unnecessary copying of memory blocks when messages move up and down a protocol stack. In particular, `writeCode` and `readCode` are used to implement code mobility and rely on the sub-package `mobility` described in Section 5.

Implementation. The IMC communication framework provides TCP/IP-level binding management mechanisms: the main classes implement the TCP/IP protocol, standard marshallers, TCP/IP connection managers, as well as standard chunk managers, schedulers, and distributed naming contexts [22].

An Example

We now show how to use the IMC communication framework to implement new networking protocols. Consider the following simple protocol called “knock-knock”, for authentication between a client C and a server S:

1	**Connect Request**	$C \rightarrow S$:	**Connect**
2	**Connect Reply**	$S \rightarrow C$:	**Knock-knock**
3	**Authentication Request**	$C \rightarrow S$:	**Who’s there?**
4	**Authentication Reply**	$S \rightarrow C$:	$Challenge$ e.g. `Will`
5	**Confirmation Request**	$C \rightarrow S$:	$Challenge$ **who?** e.g. `Will` **who?**
6	**Confirmation Reply**	$S \rightarrow C$:	$Response$ e.g. `Will you let me in?` `It’s cold out here!`

This protocol can be easily implemented using the IMC communication framework over TCP/IP bindings by a set of session and protocol objects shown in Figure 1. The client first asks the user for the server host name and TCP port number. The TCP/IP stack is initialized by creating a new instance of the `TcpIpProtocol` class and a new client session identifier with the given parameters. The communication channel is established by a `bind` call on this identifier. The returned `Session_High` object can later be used to send messages over the network: the **Connect** message is first sent. The client then waits for replies from the server. The "knock-knock" client is given by:

```
public class KnockKnockClient {
 public static void main(String[] args) {

  // Ask the user for the hostname and port number to connect to
  ...
  // Instantiate TCP/IP protocol stack
  TcpIpProtocol protocol = new TcpIpProtocol(...);
  IpSessionIdentifier id = protocol.newSessionIdentifier(hostname, port);

  // Bind to remote TCP/IP session
  Session_High session = id.bind(new KnockKnockClientSession(System.in));

  // Send "CONNECT" message
  Marshaller connectMsg = IMCMarshallerFactory.newMarshaller();
  connectMsg.writeString("Connect"); session.send(connectMsg);

  // Wait for replies from the server
  ...
 }
}
```

The top-level session object in the Knock-Knock/TCP/IP stack is a `KnockKnockClientSession` which directly reads data from standard input. When a message is received from the network, the "knock-knock" session `send` method is called. The message content is then displayed on the standard output device. Any message typed on the standard input device will be forwarded to the TCP/IP session which will send it over the network:

```
class KnockKnockClientSession implements Session_Low {
  ...
  public void send(UnMarshaller message, Session_High tcp_session) {

   // Read message from Knock−Knock server
   String fromServerMsg = message.readString();
   System.out.println("KKServer: " + fromServerMsg);

   // Get user input
   System.out.print("KKClient?"); fromUserMsg = System.in.readLine();

   // Send user input to Knock−Knock server
   Marshaller toServerMsg = IMCMarshallerFactory.newMarshaller();
   toServerMsg.writeString(fromUserMsg); tcp_session.send(toServerMsg);
  }
}
```

The “knock-knock” server begins by creating a protocol graph composed of two nodes, instances of the TcpIpProtocol and KnockKnockProtocol classes. It then waits for client invocations:

```
public class KnockKnockServer {
  public static void main(String[] args) {
    ...
    // Create Knock−Knock/TCP/IP protocol stack
    TcpIpProtocol protocol = new TcpIpProtocol(...);
    protocol.newProtocolGraph().export(new KnockKnockProtocol());

    // Wait for invocations
    ...
  }
}
```

A “knock-knock” protocol object maintains a set of “knock-knock” sessions. Each session is associated with an underlying TCP/IP session in a hashtable, in order to record the path messages should follow in the protocol stack. When exporting the “knock-knock” protocol, a TCP/IP server-side session is created containing a new server socket to listen for connection requests from clients. When a client connects, the TCP/IP session reads all messages from the network and forwards them to the higher-level KnockKnockProtocol instance: the send method of that class is called, passing as a parameter the TCP/IP server session for sending back replies to the network. A new entry in the hashtable is created, associating the TCP/IP session with a new “knock-knock” session where message processing will be performed. Control is then transferred to the send method of the “knock-knock” session:

```
class KnockKnockProtocol implements Session_Low {

  Hashtable kk_sessions; // A pool of session objects
  ...
  // Send back reply to the message coming from the network
  public void send(UnMarshaller message, Session_High tcp_session) {

    // Determine the TCP/IP session to use
    KnockKnockServerSession kk_session = null;
    synchronized (this) {
      kk_session = (KnockKnockServerSession) kk_sessions.get(tcp_session);
      if (kk_session == null) {
        kk_session = new KnockKnockServerSession(this);
        kk_sessions.put(tcp_session, kk_session);
      }
    }
    // Send the reply message on the network
    kk_session.send(message, tcp_session);
  }
}
```

The “knock-knock” session simply determines the correct message to send back to the TCP/IP session, based on the message received from the client, and following the “knock-knock” protocol. The TCP/IP session then sends the message over the network:

```
class KnockKnockServerSession {
  ...
  int state = WAITING; // State of the protocol

  // Protocol messages : challenges and responses
  static String challenges[] = { "Will", ... };
  static String responses[] = { "Will you let me in, it’s cold out here!", ... };
  int ChallengeNumber;

  // Perform ‘‘knock−knock’’ protocol message processing
  public void send(UnMarshaller message, Session_High tcp_session) {

    String fromClientMsg = message.readString();
    switch(state) {
      case WAITING:
        if (fromClientMsg.equals("Connect")) {
          toClientMsg = "Knock Knock!"; state = SENT_KNOCK_KNOCK;
        } else state = ERROR;
        break;
      case SENT_KNOCK_KNOCK:
        if (fromClientMsg.equals("Who’s there?")) {
          challengerNumber = random(MAX_CHALLENGES);
          toClientMsg = challenges[challengeNumber]; state = SENT_CHALLENGE;
        } else state = ERROR;
        break;
      case SENT_CHALLENGE:
        if (fromClientMsg.equals(challenges[challengeNumber] + "who?")) {
          toClientMsg = responses[challengeNumber]; state = WAITING;
        } else state = ERROR;
        break;
      case ERROR:
        tcp_session.close(); state = WAITING;
        break;
    }

    // Send reply message on the network
    if (state == WAITING) {
      Marshaller reply = IMCMarshallerFactory.newMarshaller();
      reply.writeString(toClientMsg); tcp_session.send(reply);
    }
  }
}
```

5 Code Mobility Management

The purpose of this part of the framework is to provide the basic functionalities for code mobility. All these functionalities are implemented in the sub-package `org.mikado.imc.mobility`. This package defines the basic abstractions for code marshalling and unmarshalling and also implements the classes for handling Java byte-code mobility transparently.

Motivations

The base classes and the interfaces of this package abstract away from the low level details of the code that migrates. By redefining specific classes of the package, the framework can be adapted to deal with different code mobility frameworks. Nowadays, most of these frameworks are implemented in Java thanks to its great means and features that help in building mobile code systems. In many of these systems, the code that is actually exchanged among sites is Java byte-code itself. For this reason, the concrete classes of the framework deal with Java byte-code mobility, and provide functionalities that can be already used, without interventions, to build the code mobility part of a Java-based code mobility framework.

When code (e.g., a process or an object) is moved to a remote computer, its classes may be unknown at the destination site. It might then be necessary to make such code available for execution at remote hosts; this can be done basically in two different ways:

- *automatic* approach: the classes needed by the moved process are collected and delivered together with the process;
- *on-demand* approach: the class needed by the remote computer that received a process for execution is requested to the server that did send the process.

We follow the automatic approach because it complies better with the mobile agent paradigm: when migrating, an agent takes with it all the information that it may need for later executions. This approach respects the main aim of this sub-package, i.e., it makes the code migration details completely transparent to the programmer, so that he will not have to worry about classes movement. Our choice has also the advantage of simplifying the handling of *disconnected operations* [29]: the agent owner does not have to stay connected after sending off an agent and can connect later just to check whether his agent has terminated. This may not be possible with the on-demand approach: the server that sent the process must always be on-line in order to provide the classes needed by remote hosts. The drawback of this approach is that code that may never be used by the mobile agent or that is already provided by the remote site is also shipped; for this reason we also enable the programmer to choose whether this automatic code collection and dispatching should be enabled.

With the automatic approach, an object will be sent along with its class binary code, and with the class code of all the objects it uses. Obviously, only the code of user defined classes has to be sent, as the other code (e.g. Java class libraries and the classes of the MIKADO framework) has to be common to every application. This guarantees that classes belonging to Java standard class libraries (and to the IMC package) are not loaded from other sources (especially, the network); this would be very dangerous, since, in general, such classes have many more access privileges with respect to other classes.

Design

The package defines the empty interface `MigratingCode` that must be implemented by the classes representing a code that has to be exchanged among distributed site. This code is intended to be transmitted in a `MigratingPacket`, stored in the shape of a `byte` array:

```
public class MigratingPacket implements java.io.Serializable {
  public MigratingPacket(byte[] b) {...}
  public byte[] getObjectBytes() {...}
}
```

How a `MigratingCode` object is stored in and retrieved from a `MigratingPacket` is taken care of by the these two interfaces:

```
public interface MigratingCodeMarshaller {
  MigratingPacket marshal(MigratingCode code) throws IOException;
}

public interface MigratingCodeUnMarshaller {
  MigratingCode unmarshal(MigratingPacket p)
    throws InstantiationException, IllegalAccessException,
      ClassNotFoundException, IOException;
}
```

These marshaller objects are used also by the classes of the `protocols` package (see Section 4). In particular the `Marshaller` and `UnMarshaller` in the package `protocols` rely on instances of `MigratingCodeMarshaller` and `MigratingCodeUnMarshaller`, respectively, to deal with `MigratingPackages`.

Starting from these interfaces, the package `mobility` provides concrete classes that automatically deals with migration of Java objects together with their byte-code, and for transparently deserializing such objects by dynamically loading their transmitted byte-code. These classes are described in the following.

Java Byte-Code Mobility. All the nodes that are willing to accept code from remote sites must have a custom *class loader*: a `NodeClassLoader` supplied by this MIKADO sub-package. When a remote object or a migrating process is received from the network, before using it, the node must add the class binary data (received along with the object) to its class loader's table. Then, during the execution, whenever a class code is needed, if the class loader does not find the code in the local packages, then it can find it in its own local table of class binary data. The most important methods that concern a node willing to accept code from remote sites are `addClassBytes` to update the loader's class table, as said above, and `forceLoadClass` to bootstrap the class loader mechanism, as explained later:

```
public class NodeClassLoader extends java.lang.ClassLoader {
 public void addClassBytes(String className, byte[] classBytes) {...}
 public Class forceLoadClass(String className) {...}
}
```

The names of user defined classes can be retrieved by means of class introspection (*Java Reflection API*). Just before dispatching a process to a remote site, a recursive procedure is called for collecting all classes that are used by the process when declaring: data members, objects returned by or passed to a method/constructor, exceptions thrown by methods, inner classes, the interfaces implemented by its class, the base class of its class.

We define a base class for all objects/process that can migrate to a remote site, `JavaMigratingCode`, implementing the above mentioned interface, `MigratingCode`, that provides all the procedures for collecting the Java classes that the migrating object has to bring to the remote site. Unfortunately, Java only provides single inheritance, thus providing a base class might restrict its usability. The problem arises when dealing with threads: the interface `Runnable` in the standard Java class library could solve the above issue but requires additional programming. For this reason we make `JavaMigratingCode` a subclass of `java.lang.Thread` (with an empty `run` method), so that `JavaMigratingCode` can be extended easily by classes that are meant to be threads. Thus, the most relevant methods for the programmer are the following ones:

```
public class JavaMigratingCode extends Thread implements MigratingCode {
 public void run() { /* empty */ }
 public JavaMigratingPacket make_packet() throws IOException {...}
}
```

The programmer will redefine `run` if its class is intended to represent a thread. The method `make_packet` will be used directly by the other classes of the framework or, possibly, directly by the programmer, to build a packet containing the serialized (marshalled) version of the object that has to migrate together with all its needed byte code. Thus, this method will actually take care of all the code collection operations.

Once these class names are collected, their byte code is gathered in the first server from which the object was sent, and packed along with the object in a `JavaMigratingPacket` object (a subclass of `MigratingPacket` storing the byte-code of all the classes used by the migrating object, besides the serialized object itself). Notice that the migrating object (namely, its variables) is written in an array of bytes (inherited by `MigratingPacket`) and not in a field of type `JavaMigratingCode`. This is necessary because otherwise, when the packet is received at the remote site and read from the stream, the remote object would be deserialized and an error would be risen when any of its specific classes is needed (indeed, the class is in the packet but has not yet been read). Instead, by using our representation, we have that, first, the byte code of process classes is read from the packet and stored in the class loader table of the receiving node; then, the object is read from the byte array; when its classes are needed, the class loader finds them in its own table. Thus, when a node receives a process, after filling in the class loader's table, it can simply deserialize the process, without any need of explicit instantiation. The point here is that classes are always stored in the class loader's table, but they are linked (i.e., actually loaded) on-demand.

The byte code of the classes used by a migrating process or object is retrieved by the method `getClassBytes` of the class loader: at the server from where the object is first sent, the byte code is retrieved from the local file system, but when a process at a remote site has to be sent to another remote site, the byte code for its classes is obtained from the class loader's table of the node.

Finally, two classes, implementing the above mentioned interfaces `MigratingCodeMarshaller` and `MigratingCodeUnMarshaller`, will take care of actually marshalling and unmarshalling a `JavaMigratingPacket` containing a migrating object and its code:

```
public class JavaByteCodeMarshaller implements MigratingCodeMarshaller {...}

public class JavaByteCodeUnMarshaller implements MigratingCodeUnMarshaller {...}
```

In particular, the first one will basically rely on the method `make_packet` of `JavaMigratingCode`, while the second one will rely on `NodeClassLoader` to load the classes stored in the `JavaMigratingPacket` and then on Java serialization to actually deserialize the migrating code contained in the packet.

Now let us examine the code that recovers the object from a `JavaMigratingPacket`, in the `JavaByteCodeUnMarshaller`. As previously hinted, a site that is willing to receive a remote object must use a `NodeClassLoader` that will take care of loading the classes received with a `JavaMigratingPacket`. The Java class loading strategy works as follows: whenever a class A is needed during the execution of a program, if it is not already loaded, then the class loader that loaded the class that needs A, say B, is required to load the class A. This usually takes place in the background, and the only class loader involved is the system class loader. In our case, we have to make our `NodeClassLoader` load the classes of the packet of the migrating object. For this reason, we have to make sure that the received object (contained in the `MigratingPacket`) is actually retrieved by a local object whose class is loaded by the `NodeClassLoader`. Since this class is a local class, i.e., a class present in the local class library, we have to force it to be loaded by the `NodeClassLoader` and not by the system class loader. In particular, the sub-package `mobility` provides an interface, `MigratingCodeRecover` and a class, `MigratingCodeRecoverImpl`, for recovering objects and classes from a `MigratingPacket`. The steps to perform are: load the `MigratingCodeRecoverImpl` class through the class loader (by forcing its loading so to avoid it is loaded by the system class loader) and recover the received packet through the `MigratingCodeRecoverImpl` instance:

```
NodeClassLoader classloader = class_loader_factory.createNodeClassLoader();
String recover_name =
  "org.mikado.imc.mobility.MigratingCodeRecoverImpl";
MigratingCodeRecover recover =
  (MigratingCodeRecover) (classloader.forceLoadClass(recover_name, true).newInstance());
```

Notice that `recover` is declared as `MigratingCodeRecover` but its actual class is `MigratingCodeRecoverImpl` (which is a class implementing the interface `MigratingCodeRecover`). Indeed, the following code would generate a `ClassCastException`:

```
MigratingCodeRecoverImpl recover =
  (MigratingCodeRecoverImpl) (classloader.forceLoadClass(recover_name, true).newInstance());
```

since Java considers two classes loaded with different class loader as incompatible. In the wrong code snippet above, for instance, the class MigratingCodeRecoverImpl of the variable recover would be loaded through the system class loader, and it would be assigned an object of the same class MigratingCodeRecoverImpl, but loaded with NodeClassLoader. This is the reason why we have to assign the instance loaded by NodeClassLoader to a variable declared with a superclass of the actually loaded class.

Once this MigratingCodeRecover object is loaded through our NodeClassLoader, we can deserialize the received object with these two simple instructions:

```
recover.set_packet(pack);
MigratingCode code = recover.recover();
```

The method recover will return the object stored in the MigratingPacket and the classes needed by such object, stored in the packet, will be automatically loaded by the NodeClassLoader. We would like to point out that not all the classes of the received object are necessarily loaded immediately; however, each time such object needs a class to be loaded, this request will be handled transparently by the NodeClassLoader. We observe that once the object is recovered from a packet, it can be used to create another packet to be sent to another site.

By default, the JavaByteCodeUnMarshaller uses a brand new class loader (through an abstract factory) for each MigratingPacket. Thus, each migrating object will be incompatible with other migrating objects, since each one of them is loaded through a different classloader. This name space separation provides a sort of isolation that helps avoiding that migrating objects coming from different sites do not interfere with each other. If this is not the desired behavior, the JavaByteCodeUnMarshaller can be initialized with a specific NodeClassLoader instance that will always be used to load every migrating object. Alternatively, the user can provide the JavaByteCodeUnMarshaller with a customized abstract factory in order to force it to use a customized NodeClassLoader for each migrating object.

Examples

Let us now show a small tutorial on how to use this sub-package for Java byte-code migrating code. First of all the classes of objects we want to migrate must be subclasses of JavaMigratingCode:

```
public class MyCode extends JavaMigratingCode {
  MyVar v = new MyVar();

  public MyRetType getFoo(MyPar p) {...}
  ...
}
```

Now an object of this class (or of one of its possible subclasses) can be sent to a remote site by creating a MigratingPacket, through a JavaByteCodeMarshaller described above. Once such a packet is created, it can be directly written into an ObjectOutputStream that, in turn, is connected, for instance, to a network output stream:

```
public class Sender {
  ...
  void sendCode(OutputStream os) throws Exception {
    MigratingCodeMarshaller marshaller = new JavaByteCodeMarshaller();
    MigratingCode code = new MyCode();
    MigratingPacket pack = marshaller.marshal(code);
    ObjectOutputStream obj_os = new ObjectOutputStream(os);
    obj_os.writeObject(pack);
    obj_os.flush();
  }
}
```

Let us observe that the act of creating a `MigratingPacket` automatically collects all the classes that `MyCode` uses, apart from creating an array of bytes representing the state of the object to migrate. Thus, the classes `MyVar`, `MyRetType` and `MyPar` are stored in the packet as well.

The site that receives a migrating object will basically perform the complementary operations: read a `MigratingPacket` from a stream (e.g., from the network) and use a `JavaByteCodeUnMarshaller` to retrieve the object from the received packet (all the operations for loading the classes will be transparent to the programmer):

```
public class Receiver {
  ...
  JavaMigratingCode receiveCode(InputStream is) throws Exception {
    MigratingCodeUnMarshaller unmarshaller = new JavaByteCodeUnMarshaller();
    ObjectInputStream obj_is = new ObjectInputStream(ss);
    MigratingPacket pack = (MigratingPacket) obj_is.readObject();
    return (JavaMigratingCode) unmarshaller.unmarshal(pack);
  }
}
```

Notice that the object retrieved from the packet is of type `JavaMigratingCode`, thus only the methods defined in that class can be used (e.g., the method `start`, inherited by `Thread`). Moreover a cast to its actual class (that in this example is `MyCode`) is not possible because that class is unknown in the receiving site and, even if it was known such cast would make the system class loader try to load the class `MyCode`; either the system class loader fails to load the class or, however, the two instances would be incompatible as explained above.

This may seem a strong limitation, but the applications that exchange code can agree on a richer interface or base class for the migrating code, say `MyMigratingProcess`, with other methods, say `m` and `n`; such class must be present in all the sites where these applications are running so that it can be loaded by the system class loader. For this reason, the class `MyMigratingProcess` must not be inserted in the `MigratingPacket`. The class `JavaMigratingCode` provides a method, `setExcludeClasses` that allows to specify which classes must not be inserted in the packet[1]. Thus, the code of the sender shown above should be changed as follows (it delivers a `MyProcess` object, where

[1] We remind that the mobility sub-package already excludes all the Java system classes and the classes of the IMC package itself.

MyProcess inherits from the common base class MyMigratingProcess that in turns derives from JavaMigratingCode):

```
public class Sender {
  ...
  void sendCode(OutputStream os) throws Exception {
    JavaMigratingCode code = new MyProcess();
    code.setExcludeClasses("mypackage.MyMigratingProcess");
    MigratingPacket pack = code.make_packet();
    ObjectOutputStream obj_os = new ObjectOutputStream(os);
    obj_os.writeObject(pack);
    obj_os.flush();
  }
}
```

The receiving code can then assign the retrieved object to a MyMigratingProcess instance and then use the richer interface of MyMigratingProcess:

```
MyMigratingProcess code = (MyMigratingProcess) unmarshaller.unmarshal();
code.m();
code.n();
```

An alternative to setExcludeClasses is the method addExcludePackage that allows to exclude a whole package (or several packages) from the set of classes that are delivered together with a migrating object. For instance, the call to setExcludeClasses above could be replaced by the following statement:

```
  code.addExcludePackage("mypackage.");
```

This allows to enforce that the whole excluded package is available on all the sites where the migrating code is dispatched to.

When extending JavaMigratingCode, there is an important detail to know in order to avoid run-time errors that would take place at remote sites and would be very hard to discover: Java Reflection API is unable to inspect local variables of methods. This implies that if a process uses a class only to declare a variable in a method, this class will not be collected and thus, when the process executes that method on a remote site, a ClassNotFoundException may be thrown. This limitation is due to the specific implementation of Java Reflection API, but it can be easily dealt with, once the programmer is aware of the problem.

6 Implementing $D\pi$ with IMC

To evaluate applicability of the components provided by IMC a small framework, called $JD\pi$, has been developed. This framework provides the runtime environment for executing programs developed using a $D\pi$ paradigm. The implementation schema is the same as the one adopted for developing Klava [5] and X-Klaim [2, 4]: like Klava is the runtime for X-Klaim so $JD\pi$ will be the runtime for $D\pi$. In the next future, a compiler will be developed to transform $D\pi$ code into Java code that relies on $JD\pi$.

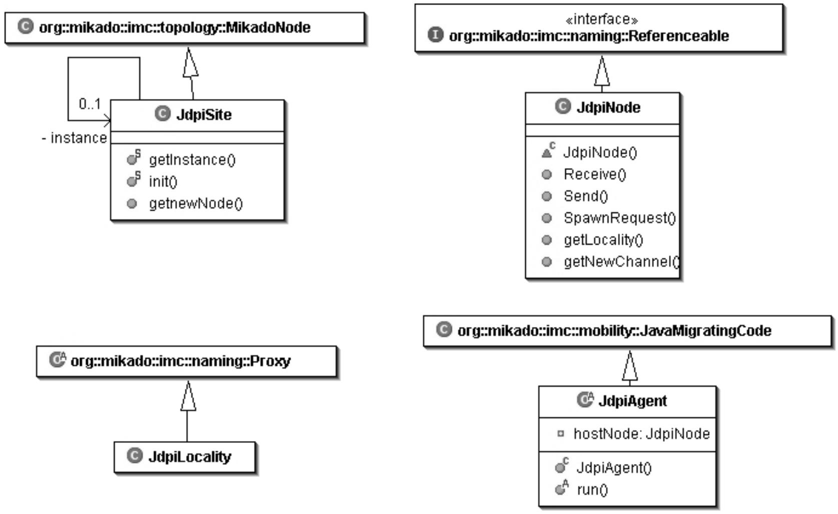


Fig. 2. Jdpi class diagrams

Design

$D\pi$, introduced by Hennessy and Riely [16], is a locality-based extension of the π-calculus [27] that requires processes to be located at nodes. More precisely the top-level consists of a parallel composition of nodes with running processes. The language is also enriched with a go primitive that permits processes to migrate to different nodes.

Analyzing the $D\pi$ paradigm, one can single out three main concepts: *Nodes*, *Processes* and *Channels*. A $D\pi$ *program* consists of a set of nodes. Each node, which is identified by a locality, contains processes running in parallel. Processes interact with each other, locally, by means of asynchronous communication performed via channels. A process can change its execution environment (the node where it is running) by performing a go l action: the execution is suspended, the process migrates at the node named l and there it restarts its computation. We assume that each host in the network may contain more $D\pi$ nodes that are executed within a common environment called *Site*.

The basic $D\pi$ ingredients are implemented by using the following classes:

- `JdpiSite`, that implements a container for nodes running on a host
- `JdpiNode`, that implements $D\pi$ nodes
- `JdpiAgent`, that implements $D\pi$ processes

Classes and Interfaces

The UML class diagram of $JD\pi$ is presented in Figure 2. In the rest of this section, we describe the classes in the diagram.

JdpiSite. JdpiSite, which extends org.mikado.imc.topology.MikadoNode class, is implemented using the pattern *singleton*. This means that only one instance of JdpiSite can be created. A reference to this instance can be obtained using the method getInstance(). JdpiSite also provides the method init() that is invoked to initialize the site. Finally a new node can be created using the method getNewNode(). JdpiSite is also responsible for providing naming services for running nodes. This is obtained at no cost, since all naming capabilities are inherited directly from MikadoNode.

JdpiNode. JdpiNode, implements a $D\pi$ node, and each instance of it runs under the control of a JdpiSite. Since processes, in order to move across the network, need to refer remotely to nodes, JdpiNode implements Referenceable. Using this approach, a process does not refer directly to a node, but it uses a JdpiLocality. In the present implementation JdpiLocality is just an abstraction for a host name and port. This class will extend org.mikado.imc.naming.Proxy (the class used to refer to remote Referenceable objects) in next development step. The class provides the method addAgent to start a new $D\pi$ process on it. Note that the only mechanism for interacting with a process running at a remote node is to spawn another process to be evaluated remotely. This is due using the method SpawnRequest(), that spawn a process to be evaluated remotely. JdpiNode is also responsible for the channel management. Methods getNewChannel(), Send() and Receive() are used, respectively to create a new channel, to send an object over a channel and to receive an object from a channel.

JdpiAgent. JdpiAgent represents a (mobile) $D\pi$ process. For this reason JdpiAgent extends JavaMigratingCode, which is defined in org.mikado.imc.mobility. The infrastructure defined in the IMC framework allow the instances of JdpiAgent to migrate from a JdpiNode to another. An instance of the JdpiAgent class does nothing by itself, acting like the nil process. Programmers need to extend it in order to implement other $D\pi$ processes. The process behavior is thus defined overriding the method execute. A JdpiAgent provides a private attribute, hostNode, that represents the hosting node. This can be used to invoke the operations over local channels, as described earlier. This attribute is set when a JdpiAgent constructor is invoked, and then only the Runtime should modify it. A JdpiAgent can also send itself to a remote node by invoking the SpawnRequest method on the hosting node. Being an extension of JavaMigratingCode, JdpiAgent extends also the Thread class. So a JdpiAgent is executed like a Java Thread, by calling the method start. Note that you can't override the run method of the Thread class. To assure that no JdpiAgent should run with a null hostNode, the run method has been declared as final and when it is called, if the hostNode is defined, it calls the execute method, otherwise it returns.

7 Conclusions

We have presented a Java package IMC that aims at providing a framework for fast prototyping distributed applications with code mobility. It aims at providing support to those building run-time systems (or virtual machines) for mobile code languages and calculi. We chose Java as the implementation platform due to its well established role in the development of this kind of software. Indeed, Java provides many useful features that are helpful in building network applications and in dynamically loading code from different sources (e.g., the network itself). However, these mechanisms still require a big programming effort, and, in this respect, they can be thought of as "low-level" mechanisms. Because of this, many existing Java based distributed systems (see, e.g., [1, 7, 8, 23, 30, 31] and the references therein) tend to implement from scratch many components that are typical and recurrent in distributed and mobile applications.

For this reason, we decided to single out the most recurrent entities of this type of applications and pack them together in a Java framework, where the architecture of distributed and mobile applications is addressed by the framework itself. The programmer can then concentrate on those parts that are really specific of his system, while relying on the framework for the recurrent standard mechanisms (node topology, communication and mobility of code). This should make the development of prototype implementations faster and should relieve the programmers from dealing with low level details. Of course, if specific applications require a specific functionality that is not in the framework (e.g., a customized communication protocol built on top of TCP/IP, or a more sophisticated mobile code management), the programmer can still customize the behaviors that concern these mechanisms in the framework.

We experimented on this matter in two respects:

- In the prototype implementation of $JD\pi$ (Section 6) we used the IMC package as it is without resorting to any customization;
- We re-engineered the implementations of our mobile code systems, TYCO and KLAVA, using the IMC package. At this stage, we had to modify/extend only specific parts of the framework (e.g., the mobility code management for TYCO and the communication protocol for KLAVA).

In both cases, we managed to concentrate our programming efforts on the main features and mechanisms of the specific distributed mobile system, and, for the rest, we relied completely on the architecture and the functionalities of the IMC framework.

Apart from the above, the Communication Protocols package was used to define customized protocol stacks by composing micro-protocols in a flexible manner. In particular, this experiment showed how new protocols can be introduced with IMC, by making evident the protocol and session objects involved, and by describing the path followed by messages within a protocol stack [21].

For the rest of the project we shall, on the one hand use the framework to implement richer languages for mobility and on the other hand we shall enrich the components to deal with security issues.

Acknowledgements. We are greatly indebted to Michele Loreti for discussions on the architecture of the framework and, especially, for suggestions about the implementations of $JD\pi$.

References

1. A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In Vitek and Tschudin [35], pages 111–130.

2. L. Bettini. *Linguistic Constructs for Object-Oriented Mobile Code Programming & their Implementations*. PhD thesis, Dip. di Matematica, Università di Siena, 2003. Available at `http://music.dsi.unifi.it`.
3. L. Bettini, M. Boreale, R. De Nicola, M. Lacoste, and V. Vasconcelos. Analysis of Distribution Structures: State of the Art. MIKADO Global Computing Project Deliverable D3.1.1, 2002.
4. L. Bettini, R. De Nicola, and R. Pugliese. X-KLAIM and KLAVA: Programming Mobile Code. In M. Lenisa and M. Miculan, editors, *TOSCA 2001*, volume 62 of *ENTCS*. Elsevier, 2001.
5. L. Bettini, R. De Nicola, and R. Pugliese. KLAVA: a Java Package for Distributed and Mobile Applications. *Software - Practice and Experience*, 32(14):1365–1394, 2002.
6. G. Boudol, I. Castellani, F. Germain, and M. Lacoste. Models of Distribution and Mobility: State of the Art. MIKADO Global Computing Project Deliverable D1.1.1, 2002.
7. G. Cabri, L. Leonardi, and F. Zambonelli. Reactive Tuple Spaces for Mobile Agent Coordination. In K. Rothermel and F. Hohl, editors, *Proc. of the 2nd Int. Workshop on Mobile Agents*, volume 1477 of *LNCS*, pages 237–248. Springer-Verlag, 1998.
8. P. Ciancarini and D. Rossi. Jada - Coordination and Communication for Java Agents. In Vitek and Tschudin [35], pages 213–228.
9. G. Cugola, C. Ghezzi, G. Picco, and G. Vigna. Analyzing Mobile Code Languages. In Vitek and Tschudin [35].
10. B. Dumant, F. Horn, F. Dang Tran, and J.-B. Stefani. Jonathan: an Open Distributed Processing Environment in Java. In *Proceedings MIDDLEWARE'98*, 1998.
11. ExoLab Group. The OpenORB project, 2002. Software available for download at `http://openorb.exolab.org/`.
12. C. Fournet and L. Maranget. The Join-Calculus Language, 1997. Software and documentation available from `http://join.inria.fr/`.
13. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
14. C. Harrison, D. Chess, and A. Kershenbaum. Mobile agents: Are they a good idea? Research Report 19887, IBM Research Division, 1994.
15. R. Hayton, A. Herbert, and D. Donaldson. Flexinet: a Flexible Component Oriented Middleware System. In *Proceedings ACM SIGOPS European Workshop*, 1998.
16. M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In U. Nestmann and B. C. Pierce, editors, *HLCL '98: High-Level Concurrent Languages (Nice, France, September 12, 1998)*, volume 16.3, pages 3–17. Elsevier Science Publishers, 1998.
17. N. Huntchinson and L. Peterson. The x-kernel: an Architecture for Implementing Network Protocols. *IEEE Transactions on Software Engineering*, 17(1):64–76, 1991.
18. R. Klefstad, D. Schmidt, and C. O'Ryan. The Design of a Real-time CORBA ORB using Real-time Java. In *Proceedings ISORC'02*, 2002.
19. F. Knabe. An overview of mobile agent programming. In *Proceedings of the Fifth LOMAPS workshop on Analysis and Verification of Multiple - Agent Languages*, number 1192 in LNCS. Springer-Verlag, 1996.
20. S. Krakowiak. *The Jonathan Tutorial: Overview, Binding, Communication, Configuration and Resource Frameworks*. ObjectWeb Consortium, 2002. Available electronically at `http://www.objectweb.org/jonathan/doc/tutorial/index.html`.
21. M. Lacoste. Building Reliable Distributed Infrastructures Revisited: a Case Study. In *International DOA Workshop on Foundations of Middleware Technologies (WFoMT'02)*, 2002.
22. M. Lacoste. IMC: Flexible Communication Support for Implementing Mobile Process Calculi. Technical report, France Telecom R&D, 2003.
23. D. Lange and M. Oshima. *Programming and Deploying Java Mobile Agents with Aglets*. Addison-Wesley, 1998.
24. F. Le Fessant. The JoCaml System Prototype, 1998. Software and documentation available from `http://join.inria.fr/jocaml`.

25. F. Levi and D. Sangiorgi. Controlling Interference in Ambients. In *Proc. 27th ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 352–364. ACM Press, 2000.
26. L. Lopes. *On the Design and Implementation of a Virtual Machine for Process Calculi*. PhD thesis, University of Porto, 1999.
27. R. Milner, J. Parrow, and J. Walker. A Calculus of Mobile Processes, I and II. *Information and Computation*, 100(1):1–40, 41–77, 1992.
28. C. O'Ryan, F. Kuhns, D. Schmidt, O. Othman, and J. Parsons. The Design and Performance of a Pluggable Protocols Framework for Real-time Distributed Object Computing Middleware. In *Proceedings MIDDLEWARE'00*, 2000.
29. A. Park and P. Reichl. Personal Disconnected Operations with Mobile Agents. In *Proc. of 3rd Workshop on Personal Wireless Communications, PWC'98*, 1998.
30. H. Peine and T. Stolpmann. The Architecture of the Ara Platform for Mobile Agents. In K. Rothermel and R. Popescu-Zeletin, editors, *Proc. of the 1st International Workshop on Mobile Agents (MA '97)*, LNCS, pages 50–61. Springer-Verlag, 1997.
31. G. Picco, A. Murphy, and G.-C. Roman. LIME: Linda Meets Mobility. In D. Garlan, editor, *Proc. of the 21st Int. Conference on Software Engineering (ICSE'99)*, pages 368–377. ACM Press, 1999.
32. D. Sangiorgi and A. Valente. A Distributed Abstract Machine for Safe Ambients. In *Proc. 28th International Colloquium on Automata, Languages and Programming (ICALP'01)*, volume 2076 of *LNCS*, pages 408–420. Springer-Verlag, 2001.
33. T. Thorn. Programming Languages for Mobile Code. *ACM Computing Surveys*, 29(3):213–239, 1997.
34. V. Vasconcelos, L. Lopes, and F. Silva. Distribution and Mobility with Lexical Scoping in Process Calculi. In *Workshop on High Level Programming Languages (HLCL'98)*, volume 16(3) of *ENTCS*, pages 19–34. Elsevier Science, 1998.
35. J. Vitek and C. Tschudin, editors. *Mobile Object Systems - Towards the Programmable Internet*, number 1222 in LNCS. Springer, 1997.
36. J. E. White. Mobile Agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press and MIT Press, 1996.

A Generic Membrane Model$^{(*)}$ (Note)

Gérard Boudol

INRIA Sophia Antipolis, BP 93 – 06902, Sophia Antipolis Cedex, France

Abstract. In this note we introduce a generic model for controlling migration in a network of distributed processes. To this end, we equip the membrane of a domain containing processes with some computing power, including in particular some specific primitives to manage the movements of entities from the inside to the outside of a domain, and conversely. We define a π-calculus instance of our model, and illustrate by means of examples its expressive power. We also discuss a possible extension of our migration model to the case of hierarchically organized domains.

1 Introduction

In the past few years, various formal models for explicit distribution and migration of code have been proposed, mostly based on the π-calculus [19] (like π_{1l} [1], Dπ, [14], *lsd*π [20], NomadicPict [23], and the Seal calculus [24]) or on the Mobile Ambients calculus [8] (like the Boxed Ambients [5] and the Safe Mobile Ambients [16]), with the notable exceptions of *OBLIQ* [6] (which is object-based), Klaim [10] (which is based on the Linda tuple space for communication and coordination), and the Distributed Join-calculus [11]. In all these models, there is a notion of a *domain* – or site, or locality [4, 9], or Ambient – where computations take place. These models also provide, more or less explicitly, primitive constructs for moving code from a domain to another. The moving entities may actually be quite diverse, from domains to closures, including processes (in the π-calculus sense), and even more so if we also consider the various programming languages, often based on Java, that have been proposed for mobile computations (see [12] for a survey).

In most of these models, the migrating entities may move quite freely from one domain to the other, provided that the destination domain is accessible, as in Dπ or the Mobile Ambients calculus for instance. However, it would clearly be better to have some means for a domain to *control* the migration of code through its boundary, for security purposes for instance. Indeed, some of the previously mentioned models offer such means. For instance, in π_{1l} [1], where failures are taken into account, the success of a migration depends on the status, running or stopped, of the destination domain. In the Seal calculus [24], moving a seal is

[(*)] Work supported by the MIKADO project of the IST-FET Global Computing Initiative.

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 208–222, 2004.

achieved by means of higher-order communication, and, as in the π-calculus, the receiver of the moving seal may be in various states, and this allows for controlling migration. Similarly, in the Safe Ambients [16], there is a channel associated with a domain for each capability, and especially **in** and **out**, and this provides a means to reject migration. However, this controlling power remains limited: for instance, one cannot in these models easily program a domain which would forward incoming entities to another destination, or dispatch them to a group of domains.

The M-calculus [22] was specifically designed to provide a notion of a *programmable domain*, in order to equip sites with various semantics, coping for instance with failures, security concerns, resource management, verification of mobile code, and so on. To this end, a domain is enriched with a new component, the controller, which in particular filters incoming and outgoing messages. The M-calculus then introduces a primitive for "passivating", or reifying a domain, which is used to derive various migration behaviours. This **passivate** construct is extremely powerful, and it may seem a little unnatural to have to reify a whole domain in order to accept an incoming agent for instance. In this paper, we shall elaborate on the idea of a programmable domain of the M-calculus, focusing on the migration aspect. In particular, we keep the idea that a domain has, besides its name and content, a controller part, which is the one we are interested in here. We call this part a *membrane*, thus elaborating on the chemical metaphor [2], where a membrane $\{\!\![\_]\!\!\}$ is used as an evaluation context, in which computations can take place. Then the main idea here is to give some "thickness" to membranes, providing them with some computing power, in order to control their *permeability*, that is to control the movements of entities between the outside and the inside of a domain (this is similar to the idea of an "airlock" in the early Cham [2]).

Our purpose is *not* to design a full programming model, aiming at Turing completeness, but rather to provide a formal model which could be combined with various programming styles regarding the content of domains. This is the sense in which our model aims at being "generic": we shall not specify any particular choice regarding the migrating entities, but we only make a few assumptions about what the content of a domain should support in order to be compatible with our membrane model. As a matter of fact, there will be only two such requirements: the content of a domain should, if migration to the outside is to be performed, be able to send messages to the enclosing membrane. Conversely, if migration to the inside is to be supported, the content of a domain should be able to dynamically accept (as new processes or threads for instance) new entities for execution. Regarding the membrane itself, that is the controller part of a domain, our model relies on a few primitives, mainly for sending messages to other domains, and adding new entities for execution to the content of the domain. Using for instance a π-calculus style for programming the membranes, we show, by means of examples, the expressive power of the proposed model.

2 The Migration Model

As indicated in the Introduction, we shall elaborate on the chemical metaphor (aka multiset rewriting, or structural equivalence), regarding a system of concurrent activities as a chemical solution (a multiset) made of molecules, possibly reacting when they come into contact. In [2] we stated a general "membrane law", asserting that computations can take place in any solution, enclosed into a membrane. In this setting, we may regard a domain as a *named membrane.* However, as we said, we will equip the membrane itself with some computing power, to control its permeability. Then we could use the notation $a\{\!\{\, S\, [\, P\,]\!\!\}$ for a domain named a, with controller S and content P. However, this turns out to be not very readable, and we will rather write $a\{S\}[P]$.

These domains with a programmable membrane are components of concurrent systems called *networks*, that may also contain "packets", that is, messages with a destination. Let us introduce some syntax for networks of domains and packets. Assuming given a set $\mathcal{N}$ of names, ranged over by n, $m\ldots$ and containing the subset $\mathcal{D}$ of domain names, ranged over by a, b, $c\ldots$, the networks are described using the following syntax:

$$A,\ B\ldots\ ::=\ a\{S\}[P]\ \mid\ a\langle M\rangle\ \mid\ (A\parallel B)\ \mid\ (\nu n)A$$

where S is any controller, P is any content, that we call a *process*, $a\langle M\rangle$ is a packet, containing a message M with destination a, $(A\parallel B)$ is the parallel composition, intended to represent the (physical or logical) juxtaposition of domains, and $(\nu n)A$ is, as in the π-calculus, scope restriction, making the name n private to the sub-network A. In this section we shall not consider any syntax for controllers S, processes P, or messages M. We shall rather assume that the controllers are states of a given transition system, and similarly for processes. We distinguish two kinds of transitions: unlabelled ones, $S \rightarrow S'$ or $P \rightarrow P'$, describe the internal computations a controller or a process may perform, which do not concern its interactions with the environment. The latter will be described by labelled transitions. Regarding the processes, we make the following assumptions:

(1) it should be possible to *dynamically add new processes* for execution to the content of a domain. We represent addition of a new process Q to an existing content P, resulting in a process P', by a labelled transition

$$P \xrightarrow{{}^{\downarrow}Q} P'$$

(2) it should be possible to *send messages* from the content of a domain to its enclosing membrane. These "upward messages" are denoted ${}^{\uparrow}M$, and the fact that the process P sends the message M to the enclosing membrane, and becomes P' in doing so, is denoted

$$P \xrightarrow{{}^{\uparrow}M} P'$$

Typically, using a CCS-like syntax for processes, we would have $P \xrightarrow{{}^{\downarrow}Q} (P \mid Q)$ and ${}^{\uparrow}M.P \xrightarrow{{}^{\uparrow}M} P$, but we shall not further analyze these transitions in the following. Regarding the membranes, controlling the migration from and to the content of a domain, we shall make similar assumptions:

(3) the membranes have the ability to receive messages (coming either from the inside or the outside of the domain), performing transitions of the form

$$S \xrightarrow{{}^{\downarrow}M} S'$$

(4) the membranes have the ability to perform specific actions to move a process from the membrane to the inside of the domain, and to send messages to another domain in the network. Moving a process from the membrane to the content of a domain is described by transitions of the form

$$S \xrightarrow{\mathbf{in}\langle P\rangle} S'$$

while sending a message in the network corresponds to a transition

$$S \xrightarrow{\mathbf{out}\langle b,M\rangle} S'$$

Let us denote by $\mathcal{S}$, $\mathcal{P}$ and $\mathcal{M}$ respectively the sets of controllers, processes and messages, and let

$$\mathcal{L}_p = \{\, {}^{\downarrow}P \mid P \in \mathcal{P}\,\} \cup \{\, {}^{\uparrow}M \mid M \in \mathcal{M}\,\}$$

and

$$\mathcal{L}_c = \{\, {}^{\downarrow}M \mid M \in \mathcal{M}\,\} \cup \{\, \mathbf{in}\langle P\rangle \mid P \in \mathcal{P}\,\} \cup \{\, \mathbf{out}\langle b, M\rangle \mid b \in \mathcal{D}, M \in \mathcal{M}\,\}$$

be respectively the sets of transition labels for processes and controllers. Then, summarizing, our migration model relies upon a system

$$\mathcal{P}roc \;=\; (\mathcal{P}, \rightarrow, \{\, \xrightarrow{L} \mid L \in \mathcal{L}_p\,\})$$

for processes, and a system

$$\mathcal{C}ontrol \;=\; (\mathcal{S}, \rightarrow, \{\, \xrightarrow{L} \mid L \in \mathcal{L}_c\,\})$$

for controllers. This is the basis on which we define the semantics of domains, and of networks. Notice that we assume here that a message is a passive entity, that does not compute by itself: there is nothing like $M \rightarrow M'$ in our model. As it has become standard (following Milner's presentation of the CHAM [17]), to formulate the operational semantics, we introduce a *structural congruence*. Assuming that a notion of free and bound occurrences of names in controllers, processes and messages is defined, as well as a notion of name substitution $\{n \mapsto m\}$, the structural congruence is the least congruence $\equiv$ on networks satisfying

$$\begin{aligned}
(\nu n)A &\equiv (\nu m)\{n \mapsto m\}A \; m \text{ not free in } A\\
((A \parallel B) \parallel C) &\equiv (A \parallel (B \parallel C))\\
(A \parallel B) &\equiv (B \parallel A)\\
((\nu n)A \parallel B) &\equiv (\nu n)(A \parallel B) \qquad n \text{ not free in } B
\end{aligned}$$

Then the transition rules are as follows. First, we have to express the fact that computations can occur within a membrane and a domain:

$$\frac{S \rightarrow S'}{a\{S\}[P] \rightarrow a\{S'\}[P]}\;(\mathrm{R1}) \qquad\qquad \frac{P \rightarrow P'}{a\{S\}[P] \rightarrow a\{S\}[P']}\;(\mathrm{R2})$$

The next rule (R3) (together with R9) asserts that a packet can be delivered to the membrane of the destination domain, if the membrane is ready to accept it. Similarly, the rule (R4) deals with the case where the message is sent from the content of the domain:

$$\frac{S \xrightarrow{\downarrow M} S'}{(a\langle M\rangle \parallel a\{S\}[P]) \rightarrow a\{S'\}[P]}\ (\mathrm{R3}) \qquad \frac{S \xrightarrow{\downarrow M} S' \quad P \xrightarrow{\uparrow M} P'}{a\{S\}[P] \rightarrow a\{S'\}[P']}\ (\mathrm{R4})$$

The rule (R5) below describes how to add a new process to the content of a domain, by means of the **in** construct, and similarly the rule (R6) describes how to send a message from a membrane to the network, by means of the **out** construct. Observe that we assume that the network is always ready to accept a message from a domain:

$$\frac{S \xrightarrow{\mathbf{in}\langle Q\rangle} S' \quad P \xrightarrow{\downarrow Q} P'}{a\{S\}[P] \rightarrow a\{S'\}[P']}\ (\mathrm{R5}) \qquad \frac{S \xrightarrow{\mathbf{out}\langle b,M\rangle} S'}{a\{S\}[P] \rightarrow (b\langle M\rangle \parallel a\{S'\}[P])}\ (\mathrm{R6})$$

Finally we have:

$$\frac{A \rightarrow A'}{(A \parallel B) \rightarrow (A' \parallel B)}\ (\mathrm{R7}) \qquad \frac{A \rightarrow A'}{(\nu n)A \rightarrow (\nu n)A'}\ (\mathrm{R8}) \qquad \frac{A \equiv B \quad A \rightarrow A'}{B \rightarrow A'}\ (\mathrm{R9})$$

Notice that a process P is not allowed to compute inside a call $\mathbf{in}\langle P\rangle$. Two important points must be noted:

(1) there is no way for a message to go directly from the outside of a domain to its content, or conversely. Any message to or from a domain has to transit through the membrane, where it is handled (rules R3-R4).

(2) there is no way for an entity to leave the membrane apart from being transmitted as an argument of the **in** and **out** primitives (R5-R6).

This means that migration, that is interactions between domains, by means of messages sent in the network, is always under the control of the membranes. The primitives **in** and **out** are predefined procedures, the semantics of which is described by (R5) and (R6), and which only have a meaning when they are called from within a membrane. Notice also that a membrane has, thanks to the **in** construct, a computing power which is of higher-order with respect to the processes running in a domain. One should also remark that our model adheres to the "*locality principle*", and more precisely to the Dπ [14] "go and communicate" philosophy: there is no reduction rule that involves more than one domain in its left-hand side. In other words, we do not require any synchronization between domains, nor do we assume any "action at a distance". In this way, we do not have to assume that the network has any particular computing power, apart from the one of routing the packets ([1]). For instance, if $S \xrightarrow{\mathbf{out}\langle b,M\rangle} S'$ and $R \xrightarrow{\downarrow M} R'$, the following transition occurs, in two steps, modulo structural congruence:

[1] and supporting scope extrusion.

$$a\{S\}[P] \parallel b\{R\}[Q] \ \xrightarrow{*}\ a\{S'\}[Q] \parallel b\{R'\}[P]$$

We could have adopted this as an atomic transition, thus doing without explicit packets $b\langle M\rangle$. However, considering this sequence as an atomic step would introduce a synchronisation between domains, whereas by our rules a message is allowed to go out of a locality even if its destination domain does not exist. We think it is more realistic to assume that the network is asynchronous – that is, sending a message is never blocked, so that the decision to choose among several messages to send is only local.

Let us further comment on our model, comparing it with the one of Dπ. In Dπ the notion of a packet is implicit: a packet $a\langle M\rangle$ is actually represented as a domain $a[M]$, and the rule for incoming messages is replaced by a structural equivalence:

$$a[P] \parallel a[Q] \equiv a[P \mid Q]$$

We do not adopt such a structural law here, because our purpose is to control incoming processes, and also because this would mean merging the membrane and the content of domains bearing the same name, and this does not fit very well with the idea of a controlled domain. Therefore the "routing" mechanism is non-deterministic, in the case where a message has two possible destinations, like in

$$a\langle M\rangle \parallel a\{S_0\}[Q_0] \parallel a\{S_1\}[Q_1]$$

which seems quite acceptable as a network behaviour. As a consequence of not having confused $a\langle M\rangle$ with $a[M]$ as in Dπ, we lose the capability one has in this calculus to dynamically create localities: in Dπ the $\mathbf{out}\langle a, M\rangle$ construct is written $a :: M$ (or $\mathbf{go}\, a.M$), where M can actually be any process, with the rule

$$b[a :: P \mid Q] \ \rightarrow\ a[P] \parallel b[Q]$$

(Thanks to the previously mentioned structural equivalence, we actually don't even have to mention Q in this rule.) It is certainly useful to have this capability of creating domains, which could be expressed with a new construct $\mathbf{mk\_dom}\langle b, S, P\rangle$, with the rule

$$\frac{S \xrightarrow{\mathbf{mk\_dom}\langle b,T,Q\rangle} S'}{a\{S\}[P] \rightarrow (a\{S'\}[P] \parallel b\{T\}[Q])}$$

However, we shall not use such a construct here. Still keeping the locality principle, a membrane could also be endowed with a capability **kill** of destroying the domain it controls, and one could also imagine some further ways in which membrane may control the content of a domain. However, we shall not explore this here.

3 A π-Calculus Based Model

In order to give more substance to our membrane model, we shall in this section introduce a particular instance, where the membranes are processes written using the π-calculus style. That is, we completely specify a *Control* system here, which essentially amounts to specifying what are the messages and how they are dealt with. This instance of our model will still be "generic", in the sense that we do not further analyze the process part $\mathcal{P}roc$. It should be easy to design a similar, Linda-based model for instance, replacing the channel based communication by communication through a tuple space, or an ML-like model, where messages are function calls (however, in this latter case we should have a way of dealing with dynamically incoming function calls), or maybe an object-oriented model, where messages are method invocations.

We assume that the set $\mathcal{N}$ of names contains a set $\mathcal{C}h$, disjoint from $\mathcal{D}$, of *channel names*, ranged over by u, v, $w\ldots$. We also sometimes regard names as *variables*, written x, y, $z\ldots$ We will not make any formal distinction between process variables, channel variables and domain variables, which respectively stand for processes, channel names and domain names (a type system would be useful for that purpose). In the examples we use $X, Y, Z\ldots$ to range over process variables. We also assume that, although they will be used in a similar way as channel names, the constants **in** and **out** of the migration model are *not* names in $\mathcal{N}$. In the π-based instance of our model, the syntax for writing control programs, occurring in the membranes, is the standard syntax of the (asynchronous, polyadic) π-calculus [3, 15, 18], enriched with the specific constructs **in** and **out** of our migration model:

$$\begin{array}{rcll}
A,\ B\ldots & ::= & a\{S\}[P] \ \mid\ a\langle M\rangle \ \mid\ (A \parallel B) \ \mid\ (\nu n)A & \textit{networks} \\
S,\ T\ldots & ::= & \mathbf{nil} \ \mid\ \mathbf{in}\langle P\rangle.S \ \mid\ \mathbf{out}\langle a, M\rangle.S \ \mid\ M & \textit{controllers} \\
 & \mid & u(\boldsymbol{x}).S \ \mid\ \mathbf{!}u(\boldsymbol{x}).S \ \mid\ (S \mid T) \ \mid\ (\nu n)S & \\
P,\ Q\ldots & ::= & \cdots & \textit{processes} \\
M & ::= & u\langle \boldsymbol{V}\rangle & \textit{messages} \\
V & ::= & n \ \mid\ P \ \mid\ \cdots & \textit{values}
\end{array}$$

The messages are tuples of values sent on a channel – which can also be interpreted as the name of a procedure, so that a message is a procedure call, with arguments. The values that can be communicated in messages are either names, or processes, or else values of the usual kind (truth values, integers...), which we freely use, together with the corresponding constructs (like conditional branching), for illustration purposes. Since processes may be arguments in a message, the controller model is of higher-order with respect to the process model. This is crucial for controlling the migration of processes, since this allows us to remove, duplicate, as well as send a process as an argument to various channels for various puposes. As usual, $u(\boldsymbol{x}).S$ is a receiver controller, waiting for some message on the channel u. Notice that we restrict replication $\mathbf{!}S$ to receivers. It should also be observed that, since we assumed that **in** and **out** are not names, there is no receiver with these names, according to our assumption that these represent predefined procedures, with a fixed – not programmable – semantics. Regarding the **in** and **out** constructs, we have adopted a "synchronous" style. That is, calling one of these procedures, in $\mathbf{in}\langle P\rangle.S$ and $\mathbf{out}\langle a, M\rangle.S$, involves an explicit continuation S. The reason is that we could not derive, as this can be done for sending names, this synchronization from an asynchronous version, where S is **nil**, because invoking **in** and **out** does not allow sending a return channel.

The operational semantics of this calculus is standard: first, we define the structural equivalence for controllers, still denoted $\equiv$, as the least equivalence compatible with the static constructs for controllers, that is

$$\frac{S \equiv S'}{(S \mid T) \equiv (S' \mid T)} \qquad \frac{S \equiv S'}{(T \mid S) \equiv (T \mid S')} \qquad \frac{S \equiv S'}{(\nu n)S \equiv (\nu n)S'}$$

and satisfying

$$\begin{aligned} (\nu n)S &\equiv (\nu m)\{n \mapsto m\}S \; m \text{ not free in } S \\ ((S \mid T) \mid U) &\equiv (S \mid (T \mid U)) \\ (S \mid T) &\equiv (T \mid S) \\ (S \mid \mathbf{nil}) &\equiv S \\ ((\nu n)S \mid T) &\equiv (\nu n)(S \mid T) \qquad n \text{ not free in } T \end{aligned}$$

Then the transition rules for controllers are the usual communication rules of the π-calculus:

$$(u\langle \boldsymbol{V} \rangle \mid u(\boldsymbol{x}).S) \to \{\boldsymbol{x} \mapsto \boldsymbol{V}\}S \qquad (u\langle \boldsymbol{V} \rangle \mid \mathbf{!}u(\boldsymbol{x}).S) \to (\{\boldsymbol{x} \mapsto \boldsymbol{V}\}S \mid \mathbf{!}u(\boldsymbol{x}).S)$$

together with

$$S \xrightarrow{\downarrow M} (S \mid M) \qquad \mathbf{in}\langle P \rangle.S \xrightarrow{\mathbf{in}\langle P \rangle} S \qquad \mathbf{out}\langle a, M \rangle.S \xrightarrow{\mathbf{out}\langle a,M \rangle} S$$

and

$$\frac{S \to S'}{(S \parallel T) \to (S' \parallel T)} \qquad \frac{S \to S'}{(\nu n)S \to (\nu n)S'} \qquad \frac{S \equiv T \quad S \to S'}{T \to S'}$$

and similarly for the labelled transitions:

$$\frac{S \xrightarrow{L} S'}{(S \parallel T) \xrightarrow{L} (S' \parallel T)} \qquad \frac{S \xrightarrow{L} S' \quad n \text{ not free in } L}{(\nu n)S \xrightarrow{L} (\nu n)S'} \qquad \frac{S \equiv T \quad S \xrightarrow{L} S'}{T \xrightarrow{L} S'}$$

This provides us with the description of the $\mathcal{C}ontrol$ system we are considering in this section (although in the examples we shall use some further primitives, as we said). Finally the structural congruence over networks has to be extended with

$$\frac{S \equiv T}{a\{S\}[P] \equiv a\{T\}[P]} \qquad \frac{n \neq a \;\&\; n \text{ not free in } P}{a\{(\nu n)S\}[P] \equiv (\nu n)a\{S\}[P]}$$

Notice that we do **not** allow communication at-a-distance, like for instance

$$a\{u\langle \boldsymbol{V} \rangle \mid S\}[P] \parallel b\{u(\boldsymbol{x}).R \mid T\}[P] \to a\{S\}[P] \parallel b\{\{\boldsymbol{x} \mapsto \boldsymbol{V}\}R \mid T\}[P]$$

This can only be achieved using $\mathbf{out}\langle b, u\langle \boldsymbol{V} \rangle\rangle$ in the a membrane. This means that the channel names, although they are "ubiquitous" and globally known, have a local meaning, provided by the receivers in the membranes.

Now let us see some examples. (We will abbreviate as usual $\mathbf{in}\langle P\rangle.\mathbf{nil}$ and $\mathbf{out}\langle a, M\rangle.\mathbf{nil}$ as $\mathbf{in}\langle P\rangle$ and $\mathbf{out}\langle a, M\rangle$ respectively.) In the π_{1l} [1] and Dπ [14] calculi there is a construction, denoted $\mathbf{go}(b, P)$ ([2]), to send the process P for execution in the remote site b. This can be interpreted as an upward message ${}^{\uparrow}\mathit{exit}\langle b, P\rangle$ to the enclosing membrane (where *exit* is a name), which is supposed to define a local protocol for sending a process elsewhere. Then, assuming that the procedure to enter a domain is named *enter*, a Dπ domain $a[P]$ can be represented ([3]) as $a\{S\}[P]$ where the membrane contains the following definitions for the ports *enter* and *exit*:

$$\begin{aligned} S = \big(\, &\mathbf{!}\mathit{enter}(X).\mathbf{in}\langle X\rangle \;\mid \\ &\mathbf{!}\mathit{exit}(y, X).\mathbf{out}\langle y, \mathit{enter}\langle X\rangle\rangle \,\big) \end{aligned}$$

For instance, if $P \xrightarrow{\mathbf{go}(b,Q)} P'$, that is $P \xrightarrow{{}^{\uparrow}\mathit{exit}\langle b,Q\rangle} P'$, and if we let $A = a\{S\}[P]$ and $A' = a\{S\}[P']$, then we have

$$A \to a\{S \mid \mathit{exit}\langle b, Q\rangle\}[P'] \to a\{\mathbf{out}\langle b, \mathit{enter}\langle Q\rangle\rangle \mid S\}[P'] \to b\langle \mathit{enter}\langle Q\rangle\rangle \parallel A'$$

Then, if $R \xrightarrow{\downarrow Q} R'$, and if we let $B = b\{S\}[R]$ and $B' = a\{S\}[R']$, we have

$$A \parallel B \xrightarrow{*} A' \parallel b\{\mathit{enter}\langle Q\rangle \mid S\}[R] \to b\{\mathbf{in}\langle Q\rangle \mid S\}[R] \to A' \parallel B'$$

Therefore this kind of membrane may be called *transparent*, since it does not perform any control on the migrating entities, and just lets them go. Notice that here, as in the following examples, we are implicitly assuming a dynamic binding mechanism: when, for instance, a process Q moving from domains to domains calls the *exit* procedure, it is the local definition, contained in the enclosing membrane, that will be executed, not the one of the site from which the process originates. This is in fact built-in in the π-calculus semantics, where names have a global, or more precisely "ubiquitous" meaning, and in the local communication discipline we have adopted (there are no distant communications, involving two distinct domains for instance).

In π_{1l} [1], a locality may fail, and is able to send messages only if it is in a "running" state. Moreover, a locality offers two public ports *stop* and *ping* respectively to make it fail and to test its status, running or stopped. Then a π_{1l} domain, initially in a "running" status, may be represented as a domain with a membrane of the following kind – assuming that some further computing constructs, like conditional branching, are available:

[2] This is written $\mathbf{spawn}(b, P)$ in π_{1l}. In Dπ this is either expressed as a message, denoted $b :: P$, or as an action prefix $\mathbf{go}\, b$.

[3] We do not claim that this provides a faithful encoding of the Dπ-calculus. Indeed, the structural equivalence $a[P] \parallel a[Q] \equiv a[P \mid Q]$ of Dπ mentioned above, by which there is only one site bearing a given name in a network, seems difficult to handle in the present setting.

$$\begin{aligned} S = (\nu s)\big(\, & s\langle\mathbf{true}\rangle \;\mid \\ & !\mathit{enter}(X).\mathbf{in}\langle X\rangle \;\mid \\ & !\mathit{exit}(y,X).s(t).\mathbf{if}\ t\ \mathbf{then}\ (\mathbf{out}\langle y, \mathit{enter}\langle X\rangle\rangle \mid s\langle t\rangle)\ \mathbf{else}\ (\mathbf{nil} \mid s\langle t\rangle) \;\mid \\ & !\mathit{stop}.s(t).s\langle\mathbf{false}\rangle \;\mid \\ & !\mathit{ping}(y,z).s(t).(\mathbf{out}\langle y, z\langle t\rangle\rangle \mid s\langle t\rangle)\,\big) \end{aligned}$$

The local state $s\langle t\rangle$ of the membrane indicates the status of the domain: running if $t = \mathbf{true}$, and failed otherwise ([4]). Notice that this controller is very much like an object, with a local state and a set of methods. As one can see, nothing can be emitted from a failed domain, since in this case the body of the *exit* procedure is equivalent to **nil**. The syntax for the *ping* construct is slightly different from the one of π_{1l}, because that calculus uses a global communication discipline, while we assume here the local communication discipline of Dπ. Specifically, a $\mathit{ping}\langle b, v\rangle$ message provides as arguments the locality b and the port v at that locality to which to send the result of the invocation of the *ping* procedure, that is the status (**true** for "running", **false** for "failed") of the locality. On the other hand, we have followed the π_{1l} formalization of a failed site, which still accepts incoming processes. We could however have an alternative semantics for the *enter* procedure, which makes the membrane of a failed site *opaque*, namely

$$!\mathit{enter}(X).s(t).\mathbf{if}\ t\ \mathbf{then}\ (\mathbf{in}\langle X\rangle \mid s\langle t\rangle)\ \mathbf{else}\ (\mathbf{nil} \mid s\langle t\rangle)$$

We would get a different semantics with an "elastic" membrane, on which messages are bouncing when the domain has failed:

$$!\mathit{enter}(X).s(t).\mathbf{if}\ t\ \mathbf{then}\ (\mathbf{in}\langle X\rangle \mid s\langle t\rangle)\ \mathbf{else}\ (\mathbf{out}\langle a, \mathit{enter}\langle X\rangle\rangle \mid s\langle t\rangle)$$

Indeed, if there are domains with the same name a, the rejected messages get a chance to be accepted somewhere else. As another example, one can imagine that the *exit* procedure, instead of sending directly its argument to its destination as in the "transparent membranes" of Dπ, sends it to a local routing procedure, defined in each node. Then the *exit* procedure would be replaced by

$$\begin{aligned} & !\mathit{exit}(y,X).\mathit{route}\langle y,X\rangle \;\mid \\ & !\mathit{route}(y,X).\mathbf{if}\ y = \mathbf{host}\ \mathbf{then}\ \mathbf{in}\langle X\rangle \\ & \qquad\qquad\qquad \mathbf{else\ let}\ z = \mathit{next\_hop}\langle y\rangle\ \mathbf{in}\ \mathbf{out}\langle z, \mathit{route}\langle y,X\rangle\rangle \end{aligned}$$

Here we assume a constant **host**, the (local) value of which is the name of the domain in which it occurs, and we also assume that a local routing table, called *next_hop*, is available, which gives the next node on the route towards the given destination.

One can easily imagine other examples, like forwarding incoming entities to a group of sites, or delegating them for processing to another site, for instance.

[4] The message $s\langle t\rangle$ is also used as a lock for the mutual exclusion of the procedures *exit*, *stop* and *ping*.

In this way, one can establish a logical hierarchy of domains, where a site only directly accepts incoming entities if they come from a given group of localities, and otherwise delegates their processing to another site. For instance, assuming that a component value transmitted in a packet is a group g of localities, a site may accept or reject migrating processes depending on a predicate p on groups:

$$!\mathit{enter}(g,X).\mathbf{if}\ p\langle g\rangle\ \mathbf{then}\ \ \mathbf{in}\langle X\rangle\ \mathbf{else\ nil}$$

Another example of a membrane in which the enter/exit protocol depends on the local state of the membrane is a *counting* membrane. To write it here we assume that the language is enriched with integers. Then, in this example, the local state of the membrane is a counter c which is incremented whenever some entity enters, and decremented when some entity exits the domain. Moreover, there is a bound n on the number of possibly entering entities. The code is, assuming that the membrane becomes opaque when the bound is reached:

$$\begin{aligned} S = (\nu c)\big(\, & c\langle 0\rangle \ \mid \\ & !\mathit{enter}(X).c(z).\mathbf{if}\ z < n\ \mathbf{then}\ (\mathbf{in}\langle X\rangle \mid c\langle z+1\rangle)\ \mathbf{else}\ (\mathbf{nil} \mid c\langle z\rangle) \ \mid \\ & !\mathit{exit}(y,X).c(z).(\mathbf{out}\langle y, \mathit{enter}\langle X\rangle\rangle \mid c\langle z-1\rangle)\,\big) \end{aligned}$$

4 On Migration in a Hierarchically Structured Network

In this section we discuss a possible adaptation of our model to a hierarchical organization of domains. In a hierarchically structured network, a domain may be embedded into another one. Since in our model a domain has two parts – the membrane and the content –, there are a priori two possibilities for the nesting of domains: a domain may be embedded into the membrane, or into the content part of another domain. However, since the membrane only has the rôle of controlling the gates of a domain, only the second possibility looks meaningful. It is not difficult to adapt our model to this case of hierarchically structured networks. However, it is clear that we cannot in this case maintain to the same extent the "parametric" aspect of our model: we have to make strong assumptions about the process language, namely that it allows for the construction of networks of domains as processes. That is, the process syntax should contain the following clauses:

$$P,\ Q\ldots\ \ ::=\ \ a\{S\}[P]\ \mid\ a\langle M\rangle\ \mid\ (P \parallel Q)\ \mid\ (\nu n)P\ \mid\ \cdots$$

so that we may represent the dynamic addition of new processes as follows, using parallel composition:

$$Q \xrightarrow{\downarrow P} (Q \parallel P)$$

Then the transition system $\mathcal{P}roc$ for processes should satisfy the rules governing the behaviour of networks, (R1) to (R9). On the other hand, there is nothing to change in the semantics of controllers, and we may use in particular the π-calculus instance of the previous section, with the same examples.

In the rest of this section we assume that the controllers are encoded as π-calculus expressions, as described in the previous section, and we discuss a possible adaptation and extension of the model in this case. With hierarchically structured domains, where domains are processes, we gain the ability of moving domains around, in an "objective" manner (see [8] for this terminology), that is as the content of messages, since a process, and therefore also a domain is a transmissible value. (We actually also have the ability to make a whole network migrate. It is not yet clear how useful this extra expressive power is, but this certainly deserves to be investigated.) This suggests that we could enrich the model, following the ideas of the Mobile Ambients calculus [8] (and of [11, 23]), with the ability for a membrane to send the domain it controls, in a "subjective" manner, as the content of a message. Since the destination of a message is either the enclosing membrane, in the case of upward messages, or a sibling domain, in the case of a packet, the reification of a domain as a message may take two forms ([5]). Moreover, we also have to provide the name of a channel to which the reified domain will be sent. To this end, we add to the model two new primitives $\mathbf{up}\langle u\rangle$ and $\mathbf{to}\langle a, u\rangle$ to the syntax of controllers.

We also notice that, since a process may now be put in parallel with a domain, the entities that may go out of a membrane, by means of the **out** construct, no longer have to be restricted to packets $a\langle M\rangle$. This means that we may now use this construct with a slightly different syntax, namely $\mathbf{out}\langle P\rangle$, so that we now write $\mathbf{out}\langle a\langle M\rangle\rangle$ instead of $\mathbf{out}\langle a, M\rangle$. It is obvious how to generalize the semantics, modifying the rule (R6):

$$\frac{S \xrightarrow{\mathbf{out}\langle Q\rangle} S'}{a\{S\}[P] \to (Q \parallel a\{S'\}[P])}\ (\mathrm{R6})'$$

This, however, should not be allowed if $a\{S\}[P]$ is a top level domain, component of a network, since a process cannot run at this level. We leave for further research the question of how to cope with this, perhaps using a type system in the style of [7]. Finally the syntax of the π-calculus based model of hierarchically structured domains is therefore as follows:

$$\begin{array}{rcll}
A,\ B\ldots & ::= & a\{S\}[P] \ \mid\ a\langle M\rangle \ \mid\ (A \parallel B) \ \mid\ (\nu n)A & \textit{networks}\\
S,\ T\ldots & ::= & \mathbf{nil} \ \mid\ \mathbf{in}\langle P\rangle.S \ \mid\ \mathbf{out}\langle P\rangle.S & \textit{controllers}\\
& \mid & \mathbf{up}\langle u\rangle.S \ \mid\ \mathbf{to}\langle a,u\rangle.S & \\
& \mid & M \ \mid\ u(\boldsymbol{x}).S \ \mid\ \mathbf{!}u(\boldsymbol{x}).S \ \mid\ (S \mid T) \ \mid\ (\nu n)S & \\
P,\ Q\ldots & ::= & a\{S\}[P] \ \mid\ a\langle M\rangle \ \mid\ {}^{\uparrow}M \ \mid\ (P \parallel Q) \ \mid\ (\nu n)P \ \mid\ \cdots & \textit{processes}\\
M & ::= & u\langle \boldsymbol{V}\rangle & \textit{messages}\\
V & ::= & n \ \mid\ P \ \mid\ \cdots & \textit{values}
\end{array}$$

We shall not repeat the semantics of the previously introduced constructs, but only give the meaning of the new primitives:

[5] If we had a π-calculus syntax for processes for instance, there would be another possibility, which is to send a domain on a channel.

$a\{\mathbf{up}\langle u\rangle.S \mid T\}[P] \rightarrow {}^{\uparrow}u\langle a\{S \mid T\}[P]\rangle \quad a\{\mathbf{to}\langle b,u\rangle.S \mid T\}[P] \rightarrow b\langle u\langle a\{S \mid T\}[P]\rangle\rangle$

Again, the first of these transitions should not be allowed if performed by a top level domain, component of a network, since an upward message ${}^{\uparrow}M$ cannot be a component of a network.

This allows us to encode subjective moves, in the style of the Mobile Ambients calculus [8], though not in an atomic way, since our model follows the locality principle. To see this, let us assume that the membranes offer the ports *enter* and *exit*, as in the examples of the previous section. Then the membrane of a domain willing to go out of the enclosing domain should contain a call $\mathbf{up}\langle \mathit{exit}\rangle$. For instance, if

$$S \;=\; (\mathbf{!}\mathit{exit}(X).\mathbf{out}\langle X\rangle \mid S')$$

and

$$T \;=\; (\mathbf{up}\langle \mathit{exit}\rangle.\mathbf{nil} \mid T')$$

then we have

$$\begin{aligned} a\{S\}[b\{T\}[Q] \parallel P] &\rightarrow a\{S\}[{}^{\uparrow}\mathit{exit}\langle b\{T'\}[Q]\rangle \parallel P] \\ &\rightarrow a\{S \mid \mathit{exit}\langle b\{T'\}[Q]\rangle\}[P] \\ &\rightarrow a\{\mathbf{out}\langle b\{T'\}[Q]\rangle \mid S\}[P] \\ &\rightarrow b\{T'\}[Q] \parallel a\{S\}[P] \end{aligned}$$

Similarly, the membrane of a domain willing to go into a sibling domain called b should contain a call $\mathbf{to}\langle b, \mathit{enter}\rangle$. For instance, if

$$S \;=\; (\mathbf{!}\mathit{enter}(X).\mathbf{in}\langle X\rangle \mid S')$$

and

$$T \;=\; (\mathbf{to}\langle b, \mathit{enter}\rangle.\mathbf{nil} \mid T')$$

then we have

$$\begin{aligned} a\{T\}[Q] \parallel b\{S\}[P] &\rightarrow b\langle \mathit{enter}\langle a\{T'\}[Q]\rangle\rangle \parallel b\{S\}[P] \\ &\rightarrow b\{\mathit{enter}\langle a\{T'\}[Q]\rangle \mid S\}[P] \\ &\rightarrow b\{S \mid \mathbf{in}\langle a\{T'\}[Q]\rangle\}[P] \\ &\rightarrow b\{S\}[P \parallel a\{T'\}[Q]] \end{aligned}$$

In this way, we can encode subjective moves à la Mobile Ambients [8]. However, the semantics is not exactly the one of Ambients' movements. For instance, since the transition

$$a\{\mathbf{to}\langle b,u\rangle.S \mid T\}[P] \rightarrow b\langle u\langle a\{S \mid T\}[P]\rangle\rangle$$

can always be performed, a domain willing to go into a sibling one may fail to do so and be stuck in a message $b\langle \mathit{enter}\langle a\{S \mid T\}[P]\rangle\rangle$ if there is actually no sibling domain with name b. As one can see, the granularity of migration is finer in the model we propose than in the Mobile Ambients calculus. Notice also that, unlike in the Mobile Ambients calculus, in our model the critical pairs (or overlapping redexes, or "interferences", following the terminology of [16]) only occur locally. More precisely, the conflicts between capabilities, and especially **up** and **to**, occur in the membrane of a domain. In particular, according to the locality principle, these conflicts do not involve several domains. Then one may hope that this form of non-determinism is simpler to deal with than the one we find in the Mobile Ambients calculus.

5 Conclusion

We have presented a formal model for explicit distribution and migration of code based on the idea of a programmable domain, that was first implemented in the M-calculus [22]. Here we focused on programming the permeability of the membrane, in order to get a simple model, which should be compatible with a wide variety of programming styles. The M-calculus has recently been simplified by Stefani into the calculus of "kells" [21]. It seems that a model like the one proposed here could be encoded into the kell calculus, representing $a\{S\}[P]$ as a pair of nested domains, where the external one, named a, contains an encoding of S, and the internal one, with a private name, contains an encoding of P. Then the kell calculus may be regarded as a low-level model, with respect to the migration calculus we introduced. However, in the kell calculus one has the ability to bypass the discipline enforced in our membrane model. Therefore, we expect that, as usual, one has means in the low-level model to break some desirable properties (that should be expressed for instance as equivalences) that the high-level model is supposed to enforce. In other words, we expect that an encoding of a membrane model, as the one we introduced, into the kell calculus would not be fully abstract. Then we think the membrane model we have presented in this preliminary note deserves to be studied for itself.

Clearly, the membranes of a domain should be given more computing power than the one we considered here. In particular, it is natural to imagine that, in order to perform some verification on the migrating code, like type checking or security checks (like in the "proof carrying code"), the membrane should be able to deal with a representation [6] $`P$ of the code of the incoming processes into some manageable data structure. A first step in this direction is taken by Hennessy et al. in their Safe Dπ-calculus [13], which also explores the idea of a "programmable membrane", from a typing point of view: the "membrane" of a domain in Safe Dπ consists in (higher-order) typed ports, through which incoming code must pass. The type checking which is performed to input incoming code, based on a sophisticated type system, is the way in which migration is controlled in Safe Dπ. It would be interesting to see how this could be expressed in an extended version of our model.

References

[1] R. Amadio, *An asynchronous model of locality, failure, and process mobility,* COORDINATION'97, Lecture Notes in Comput. Sci. 1282 (1997).

[6] sometimes called "serialization", although a more appropriate terminology would be "gödelization".

[2] G. Berry, G. Boudol, *The chemical abstract machine,* Theoretical Comput. Sci. 96 (1992) 217-248.

[3] G. Boudol, *Asynchrony and the π-calculus,* INRIA Res. Report 1702 (1992).

[4] G. Boudol, I. Castellani, M. Hennessy, A. Kiehn, *Observing localities,* Theoretical Comput. Sci. 114 (1993) 31-61.

[5] M. Bugliesi, G. Castagna, S. Crafa, *Access control for mobile agents: the calculus of Boxed Ambients,* ACM TOPLAS Vol. 26 No. 1 (2004) 57-124.

[6] L. Cardelli, *A language with distributed scope,* Computing Systems Vol. 8, No. 1 (1995) 27-59.

[7] L. Cardelli, G. Ghelli, A. Gordon, *Mobility types for mobile Ambients,* ICALP'99, Lecture Notes in Comput. Sci. 1644 (1999) 230-239.

[8] L. Cardelli, A. Gordon, *Mobile Ambients,* FoSSaCS'98, Lecture Notes in Comput. Sci. 1378 (1998) 140-155.

[9] I. Castellani, *Process Algebras with Localities,* Chapter 15 of the Handbook of Process Algebras (J. Bergstra, A. Ponse and S. Smolka, Eds), Elsevier (2001) 945-1045.

[10] R. De Nicola, G. Ferrari, R. Pugliese, *KLAIM: a kernel langage for agents interaction and mobility,* IEEE Trans. on Software Engineering Vol. 24, No. 5 (1998) 315-330.

[11] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, D. Rémy, *A calculus of mobile agents,* CONCUR'96, Lecture Notes in Comput. Sci. 1119 (1996) 406-421.

[12] A. Fuggetta, G.P. Picco, G. Vigna, *Understanding code mobility,* IEEE Trans. on Soft. Eng. Vol. 24 No. 5 (1998) 342-361.

[13] M. Hennessy, J. Rathke, N. Yoshida, *SafeDpi: a language for controlling mobile code,* Comput. Sci. Tech. Rep. 02, University of Sussex (2003).

[14] M. Hennessy, J. Riely, *Resource access control in systems of mobile agents,* Information and Computation 173 (2002) 82-120.

[15] K. Honda, M. Tokoro, *An object calculus for asynchronous communication,* ECOOP'91, Lecture Notes in Comput. Sci. 512 (1991) 133-147.

[16] F. Levi, D. Sangiorgi, *Controlling interference in Ambients,* POPL'00 (2000) 352-364.

[17] R. Milner, *Functions as processes,* Math. Struct. in Comp. Science 2 (1992) 119-141.

[18] R. Milner, *The polyadic π-calculus: a tutorial,* Technical Report ECS-LFCS-91-180, Edinburgh University (1991) Reprinted in Logic and Algebra of Specification, F. Bauer, W. Brauer and H. Schwichtenberg, Eds, Springer Verlag, 1993, 203-246.

[19] R. Milner, J. Parrow, D. Walker, *A calculus of mobile processes,* Information and Computation 100 (1992) 1-77.

[20] A. Ravara, A. Matos, V. Vasconcelos, L. Lopes, *Lexically scoping distribution: what you see is what you get,* Foundations of Global Computing Workshop, ENTCS Vol. 85 (2003).

[21] J.-B. Stefani, *A calculus of kells,* Foundations of Global Computing Workshop, Electronic Notes in Comput. Sci. Vol. 85 (2003).

[22] A. Schmitt, J.-B. Stefani, *The M-calculus: a higher-order distributed process calculus,* POPL'03 (2003) 50-61.

[23] P. Sewell, P. Wojciechowski, *Nomadic Pict: language and infrastructure design for mobile agents,* IEEE Concurrency Vol. 8 No. 2 (2000) 42-52.

[24] J. Vitek, G. Castagna, *Seal: a framework for secure mobile computations,* Workshop on Internet Programming Languages, Lecture Notes in Comput. Sci. 1686 (1999) 47-77.

A Framework for Structured Peer-to-Peer Overlay Networks*

Luc Onana Alima, Ali Ghodsi, and Seif Haridi

IMIT-Royal Institute of Technology (KTH),
Swedish Institute of Computer Science (SICS)
{onana, seif}@sics.se, aligh@imit.kth.se

Abstract. Structured peer-to-peer overlay networks have recently emerged as good candidate infrastructure for building novel large-scale and robust Internet applications in which participating peers share computing resources as equals. In the past three year, various structured peer-to-peer overlay networks have been proposed, and probably more are to come. We present a framework for understanding, analyzing and designing structured peer-to-peer overlay networks. The main objective of the paper is to provide practical guidelines for the design of structured overlay networks by identifying a fundamental element in the construction of overlay networks: the embedding of $k-$ary trees. Then, a number of effective techniques for maintaining these overlay networks are discussed. The proposed framework has been effective in the development of the DKS system, whose preliminary design appears in [2].

1 Introduction

The exponential growth of the Internet has made it possible to connect billions of machines scattered around the globe and to share computing resources such as processing power, storage and content. In order to effectively exploit these resources, the trend is to use the Internet as it was originally intended. That is, a symmetric network through which machines share resources as equals. With this in mind, a number of novel distributed systems/applications characterized by large-scale and high-dynamism of their operating environment are being built. In these distributed systems, participating peers directly share resources as equals in a peer-to-peer fashion [7, 5, 17]. We name them, peer-to-peer (P2P) systems.

The high dynamism in P2P systems is due to two reasons mainly. First, there is the need for freedom, peers should be able to join or leave the system at any time. Second, peers or the underlying communication network, which is typically the Internet, can fail at any time. To cope with this dynamism, these systems should be stabilizing, that is, despite the high-dynamism, the system should converge to legitimate configurations, without external intervention.

* This work was funded by the European project PEPITO IST-2001-32234, the European project EVERGROW IST-2004-001935, the Vinnova projects PPC and GES3 in Sweden.

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 223–249, 2004.

Peer-to-peer systems are attractive in at least two respects. First, from the user standpoint, peer-to-peer computing has a huge potential, as it reduces the need for expensive back-end servers, typically used to perform complex tasks. Moreover, the administrative costs are significantly reduced, as peer-to-peer systems are in general built on autonomous systems, without a centralized administration. Second, from the scientific perspective, peer-to-peer systems are large-scale distributed systems that involve challenging issues such as fault-tolerance, scalability and security.

The current trend in building P2P systems, consists in providing an application-independent *overlay network* as a substrate on top of which novel large-scale applications can be constructed. An overlay network is a logical network on top of one or more networks. A well-known example of such networks is the Internet. The main purpose of an overlay network is to provide effective means by which a huge amount of computing resources are linked together and accessed. And, as can be seen nowadays, various high-level distributed services can be built on top of an overlay network [6, 3, 13]. The performance of these high-level distributed services strongly depends on the properties of the underlying overlay network.

Two main design approaches can be identified for building overlay networks. On the one hand, there are *un-structured overlay networks* [14, 11], in which peers are extremely autonomous. That is, a peer joins the overlay network by connecting itself to *any* other existing peers. We say that un-structured overlay networks are built in an un-controlled fashion. Unstructured overlay networks have the advantage of providing flexibility when it comes to finding resources within the system. For instance, arbitrary queries can be handled easily. However, they provide restricted guarantees, because even if a data item were inserted into the system, there is no guarantee that it will be located when needed. Furthermore, these overlay networks tend to be inefficient, as they mainly use *flooding* for search. On the other hand, there are *structured overlay networks* [26, 23, 24, 2, 1, 19], where a peer joins the overlay network by connecting itself to some other well-defined peers, based on its logical identifier. We say that structured overlay networks are built in a controlled manner. These overlay networks provide high guarantees but have a limited query language. For example, complex queries are not supported in a "natural" way.

In this paper, our focus is on structured overlay networks [25, 2, 24]. Hence, we will use the term overlay network to mean structured overlay network. The *core service* that these overlay networks provide is a location-independent *virtual identifier based-routing*[1]. That is, given a message along with a virtual identifier *vid*, the overlay network routes the message to the ultimate destination *dest(vid)*, which is related to *vid* in a well-defined manner. We discuss the relation between *vid* and *dest(vid)* in Section 2.3.

On top of the *core service* mentioned above, a number of high-level services such as Distributed Hash Table (DHT), location-independent one-to-one commu-

[1] In [8], the term key-based routing is used in place of virtual identifier based routing.

nication (point-to-point), one-to-many communication such as broadcast [13, 10] and multicast [4], object replication and caching under various consistency models can be built.

1.1 Motivations

Since the introduction of structured overlay networks, various such systems have been proposed. A partial list includes [26, 23, 24, 2, 1, 19, 16, 18, 21], and new such systems are probably to come. Unfortunately, existing structured overlay networks are presented in a fragmented way. As a consequence of this, understanding any new such systems requires significant efforts. Furthermore, the analysis of, as well as the comparison between, overlay networks becomes difficult. And, in many cases, designing novel structured overlay networks amounts to re-inventing.

A careful analysis of most of the existing structured overlay networks reveals at least the following common characteristics:

(i) *Logarithmic Diameter.* All existing structured overlay network designs we know of strive to achieve (very) large networks with logarithmic diameter while maintaining, at each peer, a *compact routing table.* Typically, the routing table at each peer is either of logarithmic or constant size. How the logarithmic diameter is ensured varies (apparently) from one system to another. Hence, to analyze the performance (in terms of overlay hops) of any new system, significant efforts has to be re-invested. Therefore, there is a need to identify any *unifying fundamental concept* behind the achievement of logarithmic degree.

(ii) *Convergence/Stabilization.* To cope with the high dynamism, structured overlay networks are designed to be *convergent* (or *stabilizing*). That is, if ever the dynamism were ceased, the overlay network should self-configures to reach and remain in legitimate configurations. And, even if the dynamism were not stopped, the network should still route messages with acceptable performance. To achieve this convergence or stabilization property, *practical maintenance techniques are needed.*

1.2 Contributions

The motivations presented above point out two important aspects in which we contribute. We propose a small set of practical design principles for understanding, analyzing, building and maintaining structured overlay networks. The contributions of this paper is the embedding of k-ary trees[2].

To simplify the understanding of the logarithmic diameter, we propose the *embedding of k-ary trees* as the fundamental concept. Briefly and intuitively, the idea is to let each peer be the root of directed acyclic graph that spans the whole system. We conjecture that any structured overlay network that ensures logarithmic diameter under normal system operation (in a deterministic fashion),

[2] In this paper, we abuse the terminology. We often use the term tree where the term rooted DAG (Directed Acyclic Graph) should be used.

is such that each peer is the root of an embedded k-ary tree along which a form of interval/compact routing [27] is exploited.

In addition to the embedding of $k-$ary trees, we elaborate on the following techniques introduced in [2],[12]:

(ii) *Local Atomic Operation*: To reduce disturbances when peers join and leave the system, we propose that the join as well as the leave operation be managed by restricted atomic operation that involve only a small part of the overlay network.

(iii) *Correction-on-Use*: The use of local atomic operation does not guarantee that the system will "immediately" be in legitimate configurations whenever peers join or leave the system. Anyway, this cannot be achieved in an asynchronous distributed system. So, because of joins, leaves and failures, routing table at peers become inaccurate. We propose the correction-on-use technique as the basic technique for maintaining structured overlay networks. With this technique, unnecessary bandwidth consumption is avoided. And, for the system to perform optimally even under perturbation, the injected traffic should be high enough.

(iv) *Correction-on-Change*: With correction-on-change, whenever a change occurs in the system, all the peers that need to be updated are notified. The main challenge is to find, efficiently, those peers that need to be updated. Moreover, the notification should be performed in an efficient way. We provide effective mechanisms both for identification of the peers that need to be updated and for the notification.

1.3 Road-Map

The rest of this paper is organized as follows. Section 2 summarizes the steps to be considered when designing structured overlay networks. In Section 3, we present the principle for embedding $k-$ary trees in a virtual identifier space. The embedding of $k-$ary trees relies on the division of the virtual space that be either *relative* or *fixed*. Section 4 presents the relative division of the space. In section 5 we present the fixed division of the space. Section 6 is devoted to the principle of local atomic operation, for joins and leaves of peers. In Section 7, we present different techniques for maintaining structured overlay networks. Section 8 concludes the paper and points out some ongoing and future work.

2 Steps in Designing Structured Overlay Networks

The steps in designing structured overlay networks are summarized in Figure 1. An analysis of most of the existing structured overlay networks show that there are a number of key design decisions that are to be considered, when building a structured overlay network. We elaborate on these steps in this section.

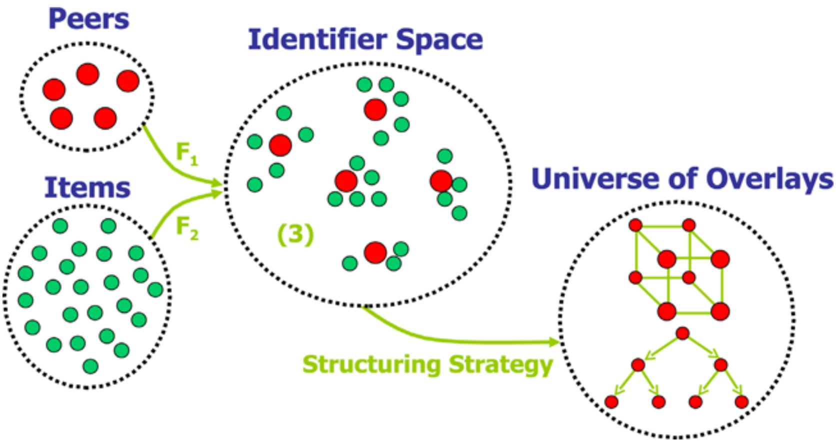


Fig. 1. a) Use F_2 to map items onto the identifier space. Use F_1 to map peers to the identifier space. Decide an assignment of items to peers. Use an embedding of $k-$ary tree to interconnect the peers

2.1 Decide an Identifier Space

Perhaps the first design decision to be made when designing a structured overlay network is to decide on what will be the the *virtual identifier space.* Throughout this document, we will use $\mathcal{I}$ to denote the identifier space. The choice of the virtual identifier space is motivated by several reasons.

1. *Addressing*: the identifier space plays the role of an address space, used for identifying resources to be interconnected by the overlay network. Each computing resource participating in a structured overlay network receives a virtual identifier taken from the virtual identifier space, $\mathcal{I}$.
2. *Scalability.* To provide access to very large sets of resources the identifier space is chosen to be a very large virtual space. This is simply an application of the well-known principle of indirection for achieving (numerical [22]) scalability, as was done for the Internet. Typically, the size $|\mathcal{I}| = N$ is in $O(k^d)$, for $k \geq 2$, and some large positive integer d. Hence for a fixed k, the size of the virtual space grows exponentially with base k.
3. *Location-Independent Communication.* Another important aspect of the virtual identifier space is that it allows peers present in the system to communicate in a point-to-point or one-to-many manner irrespective of their actual location. Thereby allowing for pure mobility.

The identifier space is assumed to have some “distance”, denoted Δ in this paper. Formally, Δ is a function of type $\Delta : \mathcal{I} \times \mathcal{I} \rightarrow \mathbb{R}$, where $\mathbb{R}$ denotes the set of real numbers. It is required that Δ satisfies at least the following three properties

(D_1) : $(\forall x, y : x, y \in \mathcal{I} : \Delta(x, y) \geq 0)$.
(D_2) : $(\forall x : x \in \mathcal{I} : \Delta(x, x) = 0)$.
(D_3) : $(\forall x, y : x, y \in \mathcal{I} : \Delta(x, y) = 0 \;\Rightarrow\; x = y)$.

and if possible, Δ can also satisfy the following two properties:

(D_4) : $(\forall x, y : x, y \in \mathcal{I} : \Delta(x, y) = \Delta(y, x))$.
(D_5) : $(\forall x, y, z : x, y, z \in \mathcal{I} : \Delta(x, z) \leq \Delta(x, y) + \Delta(y, z))$.

In case Δ satisfies all the above five properties, we have that $(\mathcal{I}, \Delta)$ is a metric space. However, in general this is not the case. In this paper, we shall say that $(\mathcal{I}, \Delta)$ is a *"pseudo-metric" space* to mean that properties $(D_1), (D_2), (D_3)$ are satisfied and possibly (D_4) and (D_5).

The closeness metric, Δ, defined on the identifier space serves two purposes:

- *clustering of resources around peers*: Typically, a resource r will be assigned to (or managed by) the peer whose virtual identifier is the closest to the virtual identifier of r. We discuss this issue in Subsection 2.3.
- *Routing message*: virtual identifiers will typically be used to route message to peers.

2.2 Mapping of Peers onto the Identifier Space

Each participating peer receives a virtual identifier, taken from $\mathcal{I}$. In Figure 1, F_1 models the mapping of peers onto the identifier space. To implement this mapping, each peer p is assumed to have some unique attribute that can be used for mapping p onto the virtual space. The implementation of F_1 can be done in several ways. One way (the typical approach) is to use a globally known hashing function such as SHA-1. The main advantage of this is that it gives a uniform distribution of peers on the identifier space. However, uniform distribution of peers over the virtual identifier space is not necessary if we want to cluster peers in some specific manner. For instance, peers can be mapped onto the virtual space in an ad-hoc fashion to achieve load-balancing or "physical" proximity.

2.3 Management of the Identifier Space by Peers

At any point in time, the identifier space is partitioned into subparts managed by peers. This is achieved by mapping each identifier in the identifier space to a set of responsible peers. Note that we here assume that an identifier can be mapped onto several peers. This is usually the case for fault-tolerance and performance improvement.

Formally, let $\mathcal{P} \subseteq \mathcal{I}$ denote the identifiers of the peers in the system at a certain point in time. A function $m_{\mathcal{P}} : \mathcal{I} \rightarrow 2^{\mathcal{P}}$ takes an identifier and returns a sub-set of the peers in the system that are responsible for that identifier.

At any point in time, each peer manages a sub-set of identifiers. The function $r_{\mathcal{P}} : \mathcal{P} \rightarrow 2^{\mathcal{I}}$ takes a peer and maps it to the sub-set of the identifier space that the peer manages. The function $r_{\mathcal{P}}$ is defined by $r_{\mathcal{P}}(p) = \{i \in \mathcal{I} | m_{\mathcal{P}}(i) = p\}$.

2.4 Mapping of Items onto the Identifier Space

As said in the introduction, an overlay network serves as the basis for interconnecting and accessing resources. Typically, it will serve for storing and retrieving data items from the system. This is achieved by mapping these items onto the virtual space.

Each resource (or item) accessible through a structured peer-to-peer overlay network receives a virtual identifier, taken from $\mathcal{I}$. In Figure 1, F_2 models the mapping of resources (items) onto the virtual identifier space.

The mapping F_2 can be implemented in several ways. The typical approach for implementing F_2 is to use a globally known hashing function. The advantage of this approach is that items are uniformly distributed on the identifier space. However, it is worth noting that the use of a hash function is not necessary. Indeed, items could be mapped such as to ensure logical proximity on the identifier space between related, or similar, items. For instance, items can be mapped onto the identifier space such that lexicographical ordering is ensured.

The advantage of a mapping that ensures logical proximity is that it enables efficient range queries for similar items. It can also be used to map items to peers within the same organizational boundary [15].

However, the disadvantage of such mappings is that they do not distribute the items uniformly over the identifier space. Therefore special care needs to be taken to ensure that the load on the peers responsible for the items is not skewed.

2.5 Decide a Structuring Strategy to Interconnect Peers

Using peer identifiers (i.e. identifiers assigned to peers) and possibly *physical proximity* like in [24, 28], an overlay network, a *directed graph*, is built. Nodes in this graph represent peers and outgoing arcs at a node of the graph model routing pointers that the peer should maintain.

Typically, a structured peer-to-peer overlay network is built such as to guarantee *logarithmic diameter* while maintaining compact routing table of logarithmic or constant size.

2.6 Decide a Strategy for Maintaining the Overlay Network

The strategy for maintaining the overlay network is an important decision step. Indeed, the techniques use for maintaining the overlay network has a significant impact on the practicality of the resulting overlay network. We think that the bandwidth will be one of the critical resources in the context of emerging peer-to-peer technologies.

When designing an overlay network, careful analysis is needed to decide on how the overlay network will be maintained. In section 7 we present effective techniques that can be used to maintain overlay networks.

3 Embedding of $k-$Ary Trees

To organize peers in an efficient overlay network, a structuring strategy that is easy to understand and implement is required.

It is a well-known fact that *logarithmic search* goes hand in hand with tree structures. This motivate our structuring strategy for connecting peers in the overlay network. We propose the *embedding of k-ary trees* in the virtual space such as to ensure overlay networks of diameter $\log_k(N)$ while maintaining, at each peer, a routing table of either logarithmic or constant size. In this paper, the focus is on the embedding of $k-$ary trees such as to maintain, at each peer, a routing table of logarithmic size. How to use our structuring strategy for overlay networks where each peer maintains a routing table of constant size is an ongoing work, however we will report some preliminary results in this paper.

The embedding of $k-$ary trees in the identifier space has several advantages. For example, it is clear that by using the virtual $k-$ary tree, the analysis of the worst, and average lookup path length becomes straightforward. Indeed, we only need to know the height as well as the arity of the embedded virtual tree. This is a simplification when compared to lengthly informal arguments often encountered in the literature.

Furthermore, the embedding of virtual trees makes it possible to introduce a novel technique for maintaining structured overlay networks. The correction-on-use technique, explained in sub-section 7.3, which serves as the basis for maintaining routing information in the DKS system. Before presenting the principle for embedding virtual $k-$ary trees in the virtual identifier space, we first introduce some assumptions and definitions.

3.1 Preliminaries

3.1.1 Assumptions

For the sake of simplicity, we assume that:

1. The virtual identifier space, denoted $\mathcal{I}$, is a discrete space organized as a ring of size N. For two arbitrary identifiers x and y, we use $x \oplus y$ (resp. $x \ominus y$) for the addition (resp. subtraction) modulo N.
2. N is a perfect power of k. That is, $N = k^d$, $k > 1$ and $d > 1$. Note that the principle described here applies as well for the case where N is not a perfect power of k.
3. In this paper the distance function $\Delta : \mathcal{I} \times \mathcal{I} \rightarrow \mathbb{N}$ is defined as follows: [3]

$$\Delta(x, y) = y \ominus x \tag{1}$$

3.1.2 Definitions

In this section the definitions used in the rest of this paper are presented.

As previously mentioned, a generic function $m_{\mathcal{P}}$ is used for the management of the identifier space. For simplicity we abuse notation and assume that the function only maps each identifier to one single peer. In this paper we will use the function $succ_{\mathcal{P}}$ as an instance of $m_{\mathcal{P}}$ to map each identifier to the peer managing the identifier.

[3] Note that the co-domain of the distance function Δ is a subset of $\mathbb{R}$, as we do not deal with real numbers.

Definition 1. *Let $succ_{\mathcal{P}} : \mathcal{I} \rightarrow \mathcal{P}$ be defined as follows.*

$$succ_{\mathcal{P}}(i) = i \oplus \min\{\Delta(i,j) \mid j \in P\} \tag{2}$$

We say that a peer $p \in \mathcal{P}$ is the successor of an identifier i iff $succ_{\mathcal{P}}(i) = p$. Given the mapping of items to the virtual identifier space (see Section 2.4), we say that each item will be managed by its successor. I.e., an item o is stored at peer $succ_{\mathcal{P}}(F_2(o))$.

Similarly to the function $succ_{\mathcal{P}}$ we define the function $pred_{\mathcal{P}}$ to denote the peer preceding a given identifier.

Definition 2. *Let $pred_{\mathcal{P}} : \mathcal{I} \rightarrow \mathcal{P}$ be defined as follows.*

$$pred_{\mathcal{P}}(i) = i \oplus \max\{\Delta(i,j) \mid j \in P\} \tag{3}$$

The function $closest_n$ takes a set of identifiers and maps it to the element in that set that is closest to n on the virtual identifier space, assuming clockwise orientation on the ring.

Definition 3. *Let $n \in \mathcal{I}$. We define $closest_n : 2^{\mathcal{I}} \rightarrow \mathcal{I}$ as follows.*

$$closest_n(Z) = n \oplus \min\{\Delta(n,j) \mid j \in Z\} \tag{4}$$

3.2 The Principle for Embedding *k*-Ary Trees

The principle is to let each element of the identifier space, n, be the root of a rooted directed acyclic graph denoted *I-DAG*(n). *I-DAG*(n) is a tree of height d that *spans* the whole identifier space and is mainly used to determine *intervals* in the identifier space. In addition, each peer, p, is the root of a rooted directed acyclic graph denoted *R-DAG*(p). Each node in the *R-DAG*(p) denotes the set of responsible peers for the corresponding intervals in *I-DAG*(p). Hence, *I-DAG*(p) and *R-DAG*(p) conceptually show, as in interval routing/compact routing, how the routing process goes for a query starting at the peer p.

The principle for embedding $k-$ary trees consists of three steps.

Step 1: From the virtual identifier space, $\mathcal{I}$, and for each identifier $p \in \mathcal{I}$, a rooted directed acyclic graph denoted *I-DAG*(p), is produced by a *systematic and recursive division of* $\mathcal{I}$. The division of the space can be done either in a *relative* or in a *fixed* fashion, as we show in Section 4 and Section 5. Each step of the division process *partitions* the *current space* into at most k sub-spaces.

Step 2: For each identifier p, a virtual $k-$ary tree of height d rooted at p is derived from *I-DAG*(p). We denote by *R-DAG*(p) the $k-$ary tree associated with identifier p.

Step 3: For each peer p, a routing table, denoted Rt_p, is derived from *R-DAG*(p) taking into account the peers present in the system. Notice that due to the dynamism, this routing table is actually time-dependent. In addition to the routing table, every peer p will maintain a pointer to the successor of its identifier

p plus one, i.e. $succ_{\mathcal{P}}(p \oplus 1)$. Similarly, each peer also maintains a pointer to its predecessor, i.e. $pred_{\mathcal{P}}(p \ominus 1)$.

The net effect of the embedding of $k-$ary trees in the identifier space is that, each participating peer will have the ability to "see" the virtual identifier space from different perspectives (in the case where each peer maintains a routing table of logarithmic size) that correspond each, to a level of its associated $k-$ary tree.

4 Relative Division of the Space

We now explain the relative space division principle. We proceed in three steps. For an identifier p, we show the construction of *I-DAG*(p). Second we show how to derive *R-DAG*(p). Then we construct Rt_p. At each step, we illustrate the concept by examples. Our approach for the relative division of the space was initially presented in [9, 2]. In the present paper, we further formalize the principle.

4.1 Constructing *I-DAG*(p)

In the relative division of the space, each identifier p has an associated rooted directed acyclic graph *I-DAG*(p) that spans the whole identifier space. The directed acyclic graph of an identifier p is different from the directed acyclic graph of any other identifier q. Hence, formally, one key invariant of the relative division of the space is

$$(\forall p, q \in \mathcal{I} :\ p \neq q\ : \textit{I-DAG}(p) \neq \textit{I-DAG}(q))$$

For an identifier n, each node in the *I-DAG*(n) is a sub-set of $\mathcal{I}$. The root of *I-DAG*(n) is $\mathcal{I}$, that is the whole set of identifiers.

To derive all the nodes of *I-DAG*(n), the division process takes the root of *I-DAG*(n) as input and then partitions it into k sub-sets. This is the first step of the division. Each sub-set produced by the first step of the division is in its turn partitioned into k sub-sets. This process repeats until we reach singleton sub-sets.

The *I-DAG*(n) of an identifier n has $d+1$ levels. The root node, node at level 0, is denoted $D_0^0(n)$. At a level l $(1 \leq l \leq d)$, *I-DAG*(n) has k^l nodes, denoted $D_{i_l}^l(n)$, where $0 \leq i_l \leq k^l - 1$. These nodes are defined by:

$$D_{i_l}^l(n) = \begin{cases} \mathcal{I} & \text{if } l = 0 \wedge i_0 = 0 \\ \{j \in D_{\lfloor \frac{i_l}{k} \rfloor}^{l-1}(n)\ :\ j = n \oplus i_l k^{d-l} \oplus q, 0 \leq q \leq k^{d-l} - 1\} & \text{otherwise} \end{cases} \tag{5}$$

From (5), one can see that the node $D_{\lfloor \frac{i_l}{k} \rfloor}^{l-1}(n)$ is the parent of nodes $D_{i_l+c}^l(n)$ where $0 \leq c \leq k-1$. Formally, *I-DAG*$(n) = \big(\mathcal{V}_D, \mathcal{E}_D\big)$, where:

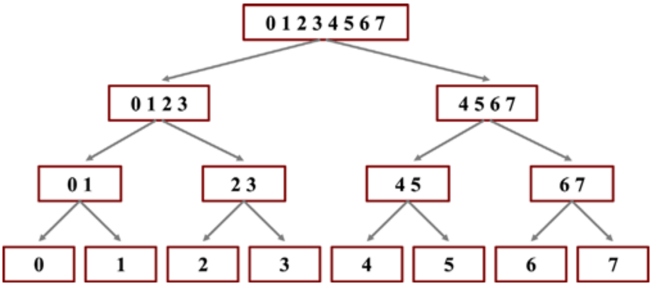


Fig. 2. Division of the space relative to identifier 0

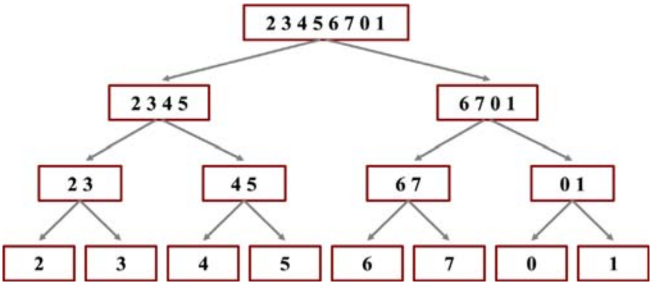


Fig. 3. Division of the space relative to identifier 2

$$\begin{aligned} \mathcal{V}_D &= \{D^l_{i_l}(p) \mid 0 \leq l \leq d, 0 \leq i_l \leq k^l - 1\} \\ \mathcal{E}_D &= \{(D^{l-1}_{\lfloor \frac{i_l}{k} \rfloor}(p), D^l_{i_l+c}(p)) \mid 1 \leq l \leq d, 0 \leq c \leq k-1\} \end{aligned} \tag{6}$$

4.1.1 The Relative Division of the Space Illustrated.

To highlight the relative aspect of the division of the space explained here, we show how the space is systematically divided for three different identifiers, namely identifier 0, 2 and 4 assuming an identifier space of size $N = 2^3$.

Given that $N = 2^3$, the space is relatively divided in three steps until subsets consisting each of a single element are reached. Figure 2 shows *I-DAG*(0), Figure 3 gives *I-DAG*(2) and Figure 4 depicts *I-DAG*(4).

One can observe from Figure 2, Figure 3 and Figure 4 that the division of the space for each identifier is different from the division of the space of any other identifier.

4.2 Deriving *R-DAG*(*p*)

From the *I-DAG*(p) one can obtain a labeled tree *R-DAG*(p) associated to p as we describe in this sub-section. For simplicity, we will assume that each node in the virtual $k-$ary tree represents a responsible peer, but the model can easily

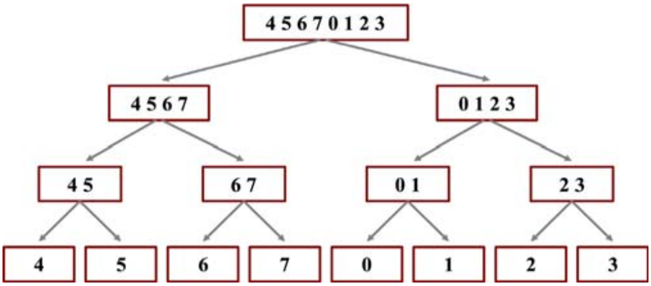


Fig. 4. Division of the space relative to identifier 4

be extended such as to have at each node of *R-DAG*(p) a sub-set of responsible peers.

The *R-DAG*(p) for an arbitrary peer $p \in \mathcal{I}$ has $d+1$ levels. The root node, node at level 0, is denoted $T_0^0(p)$. At a level l ($1 \leq l \leq d$), *R-DAG*(p) has k^l nodes, denoted $T_{i_l}^l(p)$, where $0 \leq i_l \leq k^l - 1$. These nodes are formally defined by:

$$T_{i_l}^l(p) = \begin{cases} p & \text{if } l = 0 \wedge i_0 = 0 \\ succ(closest_\phi(D_{i_l \ mod \ k}^l(\phi))) \ \text{ with } \ \phi = T_{\lfloor i_l/k \rfloor}^{l-1}(p) & \text{otherwise} \end{cases} \tag{7}$$

From (7), one can see that the node $T_{\lfloor \frac{i_l}{k} \rfloor}^{l-1}(p)$ is the parent of nodes $T_{i_l+c}^l(p)$ where $0 \leq c \leq k-1$. Formally, *R-DAG*(p) $= \big(\mathcal{V}_T, \mathcal{E}_T\big)$, where:

$$\begin{aligned} \mathcal{V}_T &= \{T_{i_l}^l(p) \mid 0 \leq l \leq d, 0 \leq i_l \leq k^l - 1\} \\ \mathcal{E}_T &= \{(T_{\lfloor \frac{i_l}{k} \rfloor}^{l-1}(p), T_{i_l+c}^l(p)) \mid 1 \leq l \leq d, 0 \leq c \leq k-1\} \end{aligned} \tag{8}$$

4.2.1 The *R-DAG*(*p*) for the Relative Division of the Space Illustrated

As an example, Figure 8 shows the virtual 3−ary tree (actually a DAG) associated to peer identified by 5 in a system where $\mathcal{P} = \{1, 2, 3, 5, 8\}$.

4.3 Deriving Routing Tables

The main goal of the division of the space is to ease the understanding of the construction of structured overlay networks. We now show how to build the routing table for an arbitrary peer p assuming that the virtual $k-$ary tree, *R-DAG*(p), associated to a peer p is known.

To build the routing table for a peer p, informally, the principle is to move from the root node of the virtual $k-$ary tree, *R-DAG*(p) associated to a peer p, down to the leaf node, $T_0^l(p)$, $0 \leq l \leq d$. At each step of the progress towards

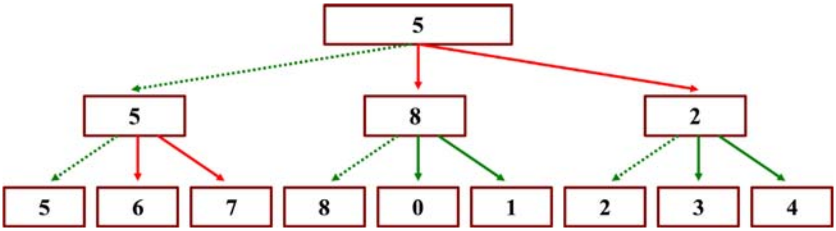


Fig. 5. Virtual 3−ary tree associated to peer 5 in a fully populated system

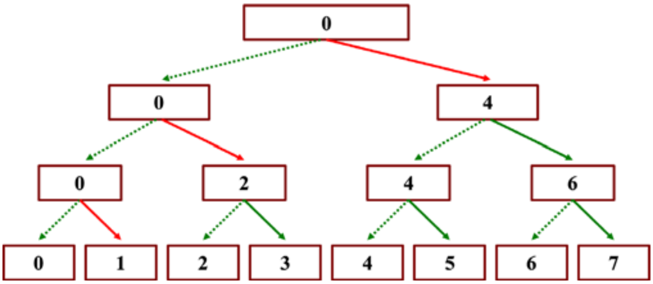


Fig. 6. Virtual 2−ary tree associated to peer 0 in a fully populated system

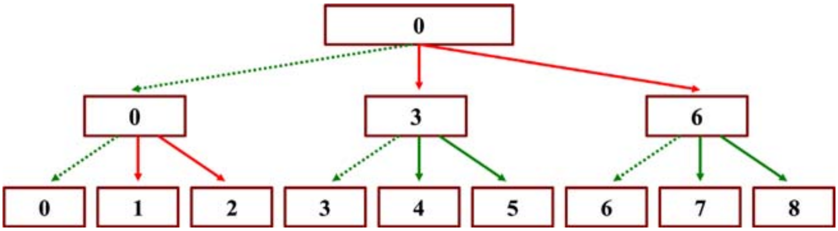


Fig. 7. Virtual 3−ary tree associated to peer 0 in a fully-populated

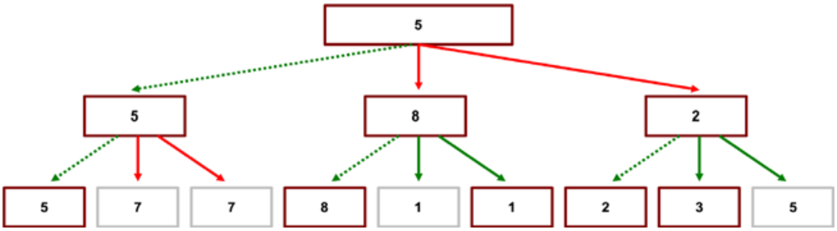


Fig. 8. Virtual 3−ary tree associated to peer 5 in a system with peers $\{1, 2, 3, 5, 8\}$

Level	Interval	Responsible
1	$[0 \cdots 3]$	0
	$[4 \cdots 7]$	4
2	$[0 \cdots 1]$	0
	$[2 \cdots 3]$	2
3	$[0]$	0
	$[1]$	1

Fig. 9. A possible routing table for peer 0 in a relative division of the space

the leaf node, $T_0^d(p)$, a pointer is maintained[4] to the responsible peer in each of $T_0^l(p)$'s immediate k children. Note that one of the pointers will always be to peer p itself, as the responsible peer for $T_0^l(p)$ is always p.

Formally, let $\mathcal{L} = \{1, \cdots, d\}$, $\mathcal{K} = \{0, \cdots, k-1\}$. The routing table Rt_p of a peer p is a function of type $\mathcal{L} \times \mathcal{K} \rightarrow \mathcal{P}$ defined as $Rt_p(l,i) = T_i^l(p)$.

Obviously, the size of the resulting routing table at a peer p is $(k-1)d$. The factor $(k-1)$ in this expression is due to that at each level $l \in \mathcal{L}$ peer p has a pointer to itself.

4.3.1 Routing Tables for the Relative Division of the Space Illustrated

As an example, consider the virtual $2-$ary tree for identifier 0 given in Figure 6, where $k = 2$ and we compute the routing table of peer 0. Moving from the root node (level 0) to level 1, pointers are stored to all the immediate children of $T_0^0(0)$: peer 0 and peer 4. From level 1 to level 2, pointers are kept to all immediate children of $T_0^1(0)$: peer 0 and peer 2. Finally, when moving from level 2 to level 3, pointers are maintained to peer 0 and peer 1.

From Figure 6, the routing table of peer 0 is immediately obtained by applying the above described principle and Figure 9 depicts the resulting routing table for peer 0. The routing table for any other peer can be computed in a similar way. For each responsible peer $Rt_0(l,i)$ we have shown the corresponding interval $D_i^l(0)$ of *I-DAG*(0). The routing process is simply intervall routing.

The reader familiar with systems such as Chord [26, 25] and DKS[2] can observe that these systems fit the relative division of the space. Indeed, these systems use the same rule as described above for choosing routing entries.

5 Fixed Division of the Space

In this section, we describe the fixed division of the space for the embedding of the $k-$ary trees. The idea behind the *fixed division* of the space is inspired from *decoding trees*, and is probably the foundation of most of the structured overlay networks which use *prefix*-based routing. Recall that decoding trees are rooted

[4] By maintaining a pointer to another peer we mean that routing information, such as network address, about the peer is maintained in a routing table.

trees used to convert sequence of code symbols to entities those code sequence represent.

As for the case of relative division of the identifier space, we proceed in three steps. First we build the *I-DAG* that result from the division of the space. Thereafter we derive the virtual $k-$ary tree, *R-DAG*(p) for a peer p. Finally, we show how to construct the routing tables for a peer.

5.1 Constructing I-DAG(p)

In the *fixed division* of the space, we build a fixed *I-DAG* from which all $k-$ary trees are derived. In contrast to the relative division of the space, the *I-DAG* is not relative to a peer, but instead fixed. Hence, we have the same *I-DAG* for all identifiers: $(\forall p, q \in \mathcal{I} : \textit{I-DAG}(p) = \textit{I-DAG}(q))$. Consequently use *I-DAG* instead of *I-DAG*(p).

In similarity with the relative division of the space, each node in the *I-DAG* will contain a set of identifiers. However, the identifiers in the fixed division of the space will be represented as strings of length d made up of symbols from the alphabet $\Sigma = \{0, .., k-1\}$ [5]

The *I-DAG* constructed for the fixed space division has the invariant that all the strings in a node of the *I-DAG* have the same prefix. More specifically, all strings of a node at a level l ($l \geq 1$) of the *I-DAG*, share a prefix of length l symbols.

We first set up some notations. In the following of this section, we use Σ^d to denote the set of all (non-empty) strings of length d, formed by concatenating symbols from Σ. The identifier space is regarded as the set Σ^d.

The root of the *I-DAG* is denoted Q_0^0 with $Q_0^0 = \Sigma^d$. That is, the root node of the fixed *I-DAG* contains all the identifiers. To derive all the other nodes of the *I-DAG*, the division process takes the root of *I-DAG* as input and then partitions it into k sub-sets. This is the first step of the division. Each sub-set produced by the first step of the division is in its turn partitioned into k sub-sets. This process repeats until we reach singleton sub-sets.

Formally, the *I-DAG* has $d+1$ levels. At a level l ($1 \leq l \leq d$), *I-DAG* has k^l nodes, denoted $Q_{i_l}^l$, where $0 \leq i_l \leq k^l - 1$. These nodes are defined by:

$$Q_{i_l}^l = \{\delta_1..\delta_d \in Q_{\lfloor \frac{i_l}{k} \rfloor}^{l-1} \mid \delta_l = i_l \bmod k\} \tag{9}$$

From (9) one can see that the node $Q_{\lfloor \frac{i_l}{k} \rfloor}^{l-1}$ is the parent of nodes $Q_{i_l+c}^l$ where $0 \leq c \leq k-1$. Formally, *I-DAG* $= \big(\mathcal{V}_Q, \mathcal{E}_Q\big)$, where:

$$\begin{aligned} \mathcal{V}_Q &= \{Q_{i_l}^l \mid 0 \leq l \leq d, 0 \leq i_l \leq k^l - 1\} \\ \mathcal{E}_Q &= \{(Q_{\lfloor \frac{i_l}{k} \rfloor}^{l-1}, Q_{i_l+c}^l) \mid 1 \leq l \leq d, 0 \leq c \leq k-1\} \end{aligned} \tag{10}$$

[5] Note that we will interchangeably use the string notation and identifier notation when convenient.

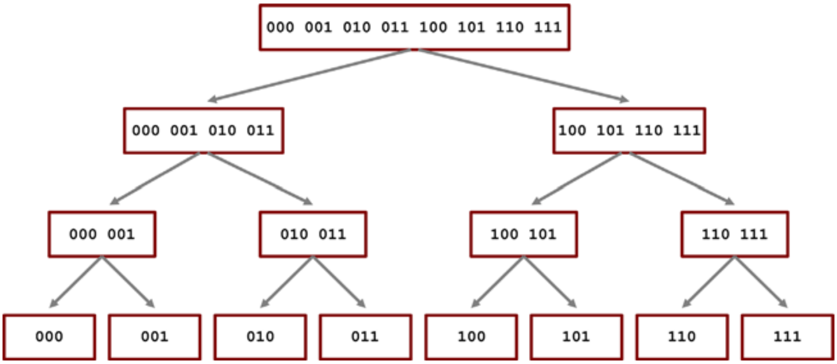


Fig. 10. Fixed division of the space for $k = 2$, $N = k^3$, $\Sigma = \{0, 1\}$

5.1.1 Fixed Division of the Space Illustrated

To illustrate the fixed division of the space, assume for the sake of simplicity, that we have an identifier space of size $N = 2^3$. We regard this virtual space as a set of strings of bits of length 3 (i.e. from 000 to 111 in this example)[6]. Then, the space is systematically divided into k (in this illustration $k = 2$) parts at each step. The result of this systematic division of the space is shown by the *I-DAG* depicted in Figure 10.

Some comments are in order regarding the fixed *I-DAG* given in Figure 10. At level 1 of this *I-DAG*, each peer in the system has the view that the whole identifier space is divided into Q_0^1 and Q_1^1, where Q_0^1 consists of all the strings of length 3, that start with the symbol 0; and Q_1^1 consists of all strings of length 3 that start with the symbol 1. So, Q_0^1 and Q_1^1 are the same for every peer present in the system. Note that this is not the case for the relative division of the space.

At level 2, all the peers in one of the sub-sets that result from the first division, have the same view of the composition of the sub-set they belong to. For example, all the peers in Q_0^1 have the same view that Q_0^1 consists of Q_0^2 and Q_1^2, where Q_0^2 is the sub-set of all the identifiers of the form $00b_3$, $b_3 \in \{0, 1\}$ and Q_1^2 is the sub-set of all the identifiers of the form $01b_3$, $b_3 \in \{0, 1\}$. By repeating the above reasoning, we obtain that at level 3, all the sub-sets that consists of single element each. More precisely, we have $Q_0^3 = \{000\}$, $Q_1^3 = \{001\}$, $Q_2^3 = \{010\}$, $Q_3^3 = \{011\}$, $Q_4^3 = \{100\}$, $Q_5^3 = \{101\}$, $Q_6^3 = \{110\}$, and $Q_7^3 = \{111\}$.

5.2 Deriving *R-DAG*(*p*)

In this section we present two alternative ways of building the virtual $k-$ary tree, *R-DAG*(p) for a peer p. We name the first *logarithmic-degree R-DAG*(p) as the size of the routing tables that are derived from it are in logarithmic order of the number of peers in the system. We name the second *constant-degree R-DAG*(p) as the size of the routing tables that are derived from it are of a constant size.

[6] That is, the code symbols are taken from the alphabet $\{0, 1\}$

5.2.1 Deriving Logarithmic Degree *R-DAG*(p)

The logarithmic-degree *R-DAG*(p) for a peer p is built such as to ensure that one can reach any peer from p in *at most* $\log_k(N)$ hops, while maintaining a routing table of size in $O(\log_k(N))$. We first give the general principle, then we illustrate by some examples.

Recall the definition of *R-DAG*(p), associated to a peer p, given in subsection 3.2.

The logarithmic-degree *R-DAG*(p) for an arbitrary peer $p \in \mathcal{I}$ has $d+1$ levels. The root node, node at level 0, is denoted $L_0^0(p)$. At a level l $(1 \le l \le d)$, *R-DAG*(p) has k^l nodes, denoted $L_{i_l}^l(p)$, where $0 \le i_l \le k^l - 1$.

The node $L^{l-1}_{\lfloor \frac{i_l}{k} \rfloor}(p)$ is the parent of nodes $L^l_{i_l+c}(p)$ where $0 \le c \le k-1$. Formally, *R-DAG*(p) $= \big(\mathcal{V}_L, \mathcal{E}_L\big)$, where:

$$\begin{aligned} \mathcal{V}_L &= \{L^l_{i_l}(p) \mid 0 \le l \le d, 0 \le i_l \le k^l - 1\} \\ \mathcal{E}_L &= \{(L^{l-1}_{\lfloor \frac{i_l}{k} \rfloor}(p), L^l_{i_l+c}(p)) \mid 1 \le l \le d, 0 \le c \le k-1\} \end{aligned} \tag{11}$$

We provide three alternative rules for defining $L^l_{i_l}(p), 0 \le l \le d, 0 \le i_l \le k^l - 1$ at a peer p. All of the rules have in common that $L^l_{i_l}(p) = p$ whenever $p \in Q^l_{i_l}$.

Rule (12) states that the successor of the numerically smallest identifier in node $Q^l_{i_l}$ is taken as the responsible peer for $L^l_{i_l}(p)$, for a peer p.

$$L^l_{i_l}(p) = \begin{cases} p & \text{if } p \in Q^l_{i_l} \\ succ_{\mathcal{P}}(\min(Q^l_{i_l})) & \text{otherwise} \end{cases} \tag{12}$$

The advantage of Rule (13) is that the responsible peers are chosen in a deterministic fashion. Therefore, techniques such as *correction-on-use* (see subsection 7.3) and *correction-on-change* (see sub-section 7.4) can be used. However, the disadvantage of Rule (13) is that it will lead to imbalanced traffic load. The reason for this is that the peers at level l in the virtual $k-$ary tree will have an in-degree of $\frac{1}{k^l}$. To solve this problem, we suggest the use of Rule (13), that uniformly distributes the choice of the responsible peers in a deterministic fashion.

$$L^l_{i_l}(p) = \begin{cases} p & \text{if } p \in Q^l_{i_l} \\ succ_{\mathcal{P}}(\min(Q^l_{i_l}) \oplus (p \bmod k^{d-l})) & \text{otherwise} \end{cases} \tag{13}$$

The deterministic nature of Rule (13) has the same advantages that were mentioned for Rule (12). One disadvantage of Rule (13) is that all the responsible peers are chosen in a deterministic fashion. Hence, there is no freedom of choosing responsible peers according to some proximity metric.

To relax the restriction imposed by Rule (13), we suggest the use of Rule (14). With this rule any peer whose identifier is in $Q^l_{i_l}$ can be randomly chosen as a responsible peer for $L^l_{i_l}(p)$, at a peer p, since all nodes in $Q^l_{i_l}$ share the same prefix. In case there is no peer with identifier in $Q^l_{i_l}$ the successor of the smallest identifier in $Q^l_{i_l}$ is chosen as a responsible peer.

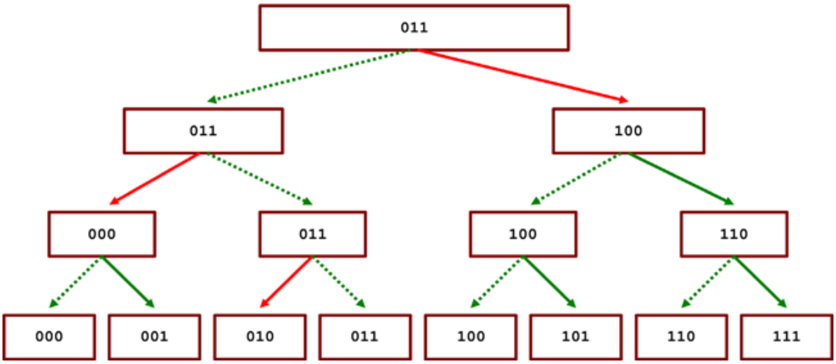


Fig. 11. *R-DAG*(011) in a fully populated system built from *I-DAG* in Figure 10 using Rule (12)

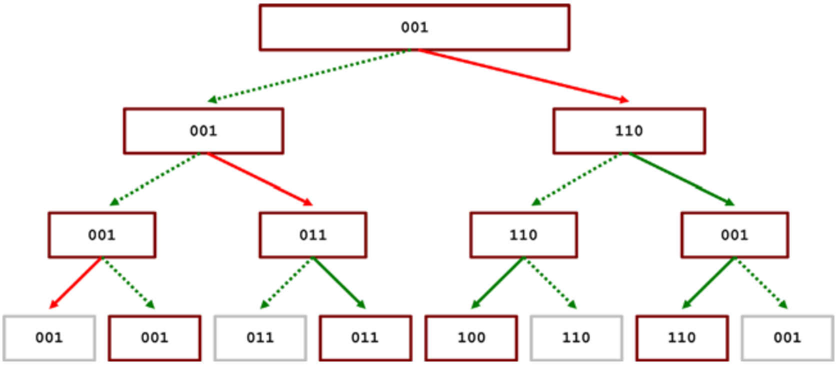


Fig. 12. *R-DAG*(001) in a system with $\mathcal{P} = \{001, 011, 100, 110\}$ built from *I-DAG* in Figure 10 using Rule (13)

The advantage of this rule is that any proximity metric, such as the round-trip time, can be used when choosing the responsible peer. The reader familiar with peer-to-peer overlay networks such as Pastry [24] and Tapstry [28] can observe that Rule (14) is exploited in these systems.

$$L^l_{i_l}(p) = \begin{cases} p & \text{if } p \in Q^l_{i_l} \\ random(Q^l_{i_l}) & \text{elseif } Q^l_{i_l} \cap \mathcal{P} \neq \emptyset \\ succ_{\mathcal{P}}(\min(Q^l_{i_l})) & \text{otherwise} \end{cases} \tag{14}$$

The Logarithmic-Degree Fixed Division of the Space Illustrated. Figure 11 shows the *R-DAG*(011) in a fully populated system built from *I-DAG* in Figure 10 using Rule (12). In Figure 12 we show *R-DAG*(001) in a system with $\mathcal{P} = \{001, 011, 100, 110\}$ built from *I-DAG* in Figure 10 using Rule (13). We illustrate the use of Rule (12) in Figure 13, in a system with $\mathcal{P} = \{001, 011, 101, 110\}$ built from *I-DAG* in Figure 10.

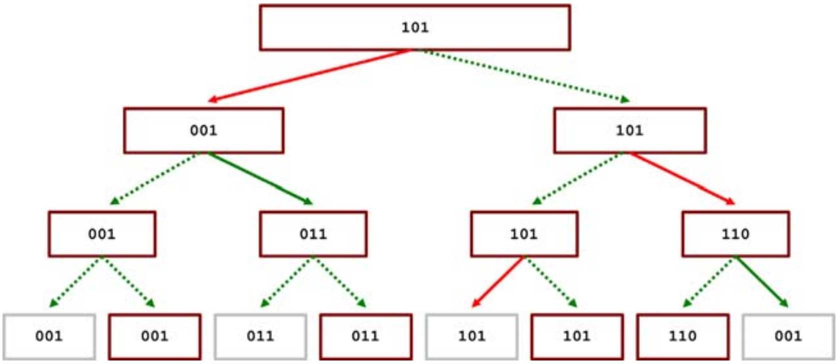


Fig. 13. *R-DAG*(101) in a system with $\mathcal{P} = \{001, 011, 101, 110\}$ built from *I-DAG* in Figure 10 using Rule (12)

5.2.2 Deriving Constant Degree *R-DAG*(*p*)

Our work on deriving constant-degree *R-DAG* is an ongoing work. We here report our preliminary result. The constant-degree *R-DAG*(p) for an arbitrary p, is built from a fixed *I-DAG*.

The constant-degree *R-DAG*(p) for an arbitrary peer $p \in \mathcal{I}$ has $d+1$ levels. The root node, node at level 0, is denoted $C_0^0(p)$. At a level l $(1 \leq l \leq d)$, *R-DAG*(p) has k^l nodes, denoted $C_{i_l}^l(p)$, where $0 \leq i_l \leq k^l - 1$.

The node $C_{\lfloor \frac{i_l}{k} \rfloor}^{l-1}(p)$ is the parent of nodes $C_{i_l+c}^l(p)$ where $0 \leq c \leq k-1$. Formally, *R-DAG*(p) $= \big(\mathcal{V}_C, \mathcal{E}_C\big)$, where:

$$\begin{aligned} \mathcal{V}_C &= \{C_{i_l}^l(p) \mid 0 \leq l \leq d, 0 \leq i_l \leq k^l - 1\} \\ \mathcal{E}_C &= \{(C_{\lfloor \frac{i_l}{k} \rfloor}^{l-1}(p), C_{i_l+c}^l(p)) \mid 1 \leq l \leq d, 0 \leq c \leq k-1\} \end{aligned} \tag{15}$$

In the following we show how $C_{i_l}^l(\delta_1 ... \delta_d)$ $(0 \leq l \leq d, 0 \leq i_l \leq k^l - 1)$ is derived from the *I-DAG* in a system where $|\mathcal{P}| = |\mathcal{I}| = N$.

$$C_{i_l}^l(\delta_1 ... \delta_d) = \begin{cases} \delta_1 ... \delta_d & \text{if } l = 0 \land i_0 = 0 \\ \{\phi_2 ... \phi_d \psi \mid \phi_1 ... \phi_d \in C_{\lfloor \frac{i_l}{k} \rfloor}^{l-1}(\delta_1 ... \delta_d), \psi = i_l \bmod k\} & \text{otherwise} \end{cases} \tag{16}$$

The Constant-Degree Fixed Division of the Space Illustrated. Figure 15 shows the *R-DAG*(21) in a fully populated system built from *I-DAG* shown in Figure 14

5.3 Deriving Routing Tables

We now show how to build the routing table for an arbitrary peer p assuming that the virtual $k-$ary tree, *R-DAG*(p), associated to a peer p is known. First, the derivation of the routing tables for the logarithmic-degree *R-DAG*(p) is shown. Thereafter the equivalent for the constant-degree *R-DAG*(p) is shown.

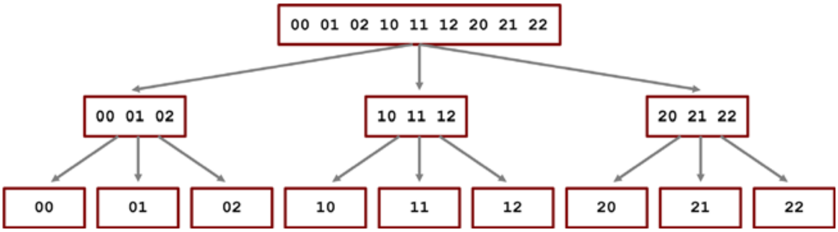


Fig. 14. *I-DAG* of a system where $d = 2$, and $\Sigma = \{0, 1, 2\}$

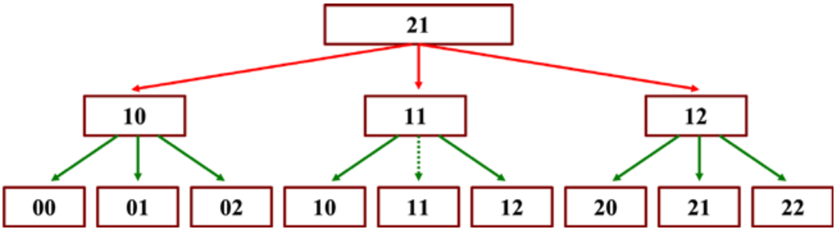


Fig. 15. *R-DAG*(21) in a fully-populated system built from *I-DAG* in Figure 14

5.3.1 Deriving the Routing Tables for the Logarithmic-Degree *R-DAG*(p)

To build the routing table for a peer p the principle is to move from the root node of *I-DAG* level by level down to the leaf node, $Q^d_{i_p}$, that contains p. Let $Q^0_{i_0} \cdots Q^d_{i_d}$ be the path from the root node Q^0_0 to the leaf node $Q^d_{i_p}$. Peer p will maintain a pointer to all the k immediate children of $L^0_{i_0}(p), \cdots, L^{d-1}_{i_{d-1}}(p)$.

Formally, let $\mathcal{L} = \{1, \cdots, d\}$, $\mathcal{K} = \{0, \cdots, k-1\}$. The routing table Rt_p of a peer p is a function of type $\mathcal{L} \times \mathcal{K} \rightarrow \mathcal{P}$ defined as $Rt_p(l, m) = L^l_{k*i_{l-1}+m}(p)$.

The size of the resulting routing table at a peer p is $(k-1)d$. The factor $(k-1)$ in this expression is due to that at each level $l \in \mathcal{L}$ peer p has a pointer to itself.

Routing Tables for the Logarithmic-Degree R-DAG(p) *Illustrated.* We will now illustrate how routing tables are constructed for the logarithmic-degree *R-DAG*(p). The routing table, Rt_{101}, is constructed for a peer 101 with the *R-DAG*(101) shown in Figure 13.

The path from the root to the leaf node containing 101 in *I-DAG* is Q^0_0, Q^1_1, Q^2_2, Q^3_5. Hence, the Rt_{101} will contain a pointer to all the k immediate children of each node $L^0_0(101)$, $L^1_1(101)$, $L^2_2(101)$. As a consequence, $Rt_{101}(1, 0) = 001$, $Rt_{101}(1, 1) = 101$, $Rt_{101}(2, 0) = 101$, $Rt_{101}(2, 1) = 110$, $Rt_{101}(3, 0) = 101$, and $Rt_{101}(3, 1) = 101$. The routing table is illustrated by Figure 16.

Level	Interval	Responsible
1	$[000 \cdots 011]$	001
	$[100 \cdots 111]$	101
2	$[100 \cdots 101]$	101
	$[110 \cdots 111]$	110
3	$[100]$	101
	$[101]$	101

Fig. 16. The routing table for peer 101 derived from *R-DAG*(101) shown in Figure 13

Level	Interval	Responsible
1	$[00 \cdots 02]$	10
	$[10 \cdots 12]$	11
	$[20 \cdots 22]$	12

Fig. 17. The routing table for peer 21 derived from *R-DAG*(21) shown in Figure 15

5.3.2 Deriving the Routing Tables for the Constant-Degree *R-DAG*(p)

In the constant-degree *R-DAG*(p) each peer will only maintain pointers to k other peers, namely all the k immediate children of the root node, $C_0^0(p)$ for a peer p.

Formally, let $\mathcal{K} = \{0, \cdots, k-1\}$. The routing table Rt_p of a peer p is a function of type $\mathcal{K} \to \mathcal{P}$ defined as $Rt_p(k) = C_k^1(p)$.

Routing Tables for the Constant-Degree R-DAG(p) *Illustrated* We now illustrate how routing tables are constructed for the constant-degree *R-DAG*(p). The routing table, Rt_{21}, is constructed for a peer 21 with the *R-DAG*(21) shown in Figure 15.

Peer 21 maintains a pointer to all the k immediate children of node $C_0^0(21)$. Consequently, $Rt_{21}(0) = 10$, $Rt_{21}(1) = 11$, $Rt_{21}(2) = 12$. The routing table is illustrated by Figure 17

6 Local Atomic Operation for Joins and Leaves

Given that the $k-$ary structuring strategy and its associated concepts of *relative* and *fixed division* of the space, we now know how to build efficient structured overlay networks. Up to now, the story is only partial, because it does not involve the joining nor the departures of peer. How to deal with joins and leaves operation such as to ensure strong guarantees? The *local atomic* principle presented in this section serves for this.

6.1 Join

We have assumed that the virtual identifier space is a ring. Hence, a join by a peer n amounts to insertion of n between two existing peers in the case of non-trivial networks, such as the one consisting with only one peer.

Due to high dynamism, several peers with consecutive identifiers might attempt to join simultaneously. So, if not done properly, such concurrent insertions might lead to undesirable situations such as *false lookup failure*, in which the system returns a message saying that an item is not present in the system while the item is actually there. Our simulation-based study of systems such as Chord [25, 26] that use very "weak" join protocol, can have significant number of false lookup failures when compared to our DKS system in which local atomic join protocol is used.

The main idea behind local atomic join is to ensure that in between any pair of peers currently in the system, there is at most one peer that is inserted at a time. This requires, assuming fault-free environment, a tight synchronization between the joining peer and the pair of peers between which the joining peer is going to be inserted. When fault-tolerance is considered, the number of peers involved in a local atomic join grows with the size of the immediate neighbors that peer maintains.

6.2 Leave

As for the join, when a peer wishes to (cooperate when) leaving the system, some synchronization is required. Otherwise, a high level of inconsistency might result.

In most structured peer-to-peer overlay networks, leaves and failures are given the same semantics. This is probably due to some simplification, as it implies that the same transition rule applies for both the leave and the failure. However, we think that these two operations should be given distinct semantics. We consider a leave operation as a "cooperative departure", and a failure is merely a "non-cooperative departure". Hence, to avoid significant amount of inconsistency when a peer leaves the system, we suggest that at any time, in between two consecutive peers, at most one peer departs. Achieving this requires the use of local atomic operation. Again, the number of involved peers depends on the level of guarantees targeted by the system.

7 Techniques for Maintaining Structured Overlay Networks

In this section, we discuss some of the existing techniques for maintaining structured overlay networks.

7.1 Periodic Stabilization

We call *periodic stabilization* the technique that consists of running, periodically, separate routines for correcting routing information that each peer maintains.

Most of the existing peer-to-peer infrastructures use this technique. For instance, it is used in systems such as Chord [25, 26], CAN [23] and Pastry [24].

The idea here is that each peer periodically checks its neighbors, to detect any change that occurs in the vicinity of the checking peer. In Chord, this is done by periodically running the *stabilize* and the *fix finger* sub-routines. This technique has the advantage that changes can be detected quickly. However, the cost of doing this periodical checking can be very high. An immediate observation that one can make is that in systems using this technique, there is an unnecessary bandwidth consumption when the system is frequently used but the dynamism in the system is low.

7.2 Adaptive Stabilization

As mentioned in the previous sub-section, periodic stabilization induces unnecessary bandwidth consumption in periods of low dynamism. To overcome this problem, an alternative approach is what we call *adaptive stabilization*, in which the rate of stabilization is tuned depending on some observed conditions or parameters, as suggested in [20]. In [20], what we here call adaptive stabilization is termed *self-tuning*, and requires some estimate of the system size and the failure rate. Intuitively, the adaptive stabilization technique might help reducing unnecessary bandwidth consumption. However, it is not yet clear what parameters are to be observed to effectively tune the probing rate. More importantly, how to make these observations is currently not well understood, given the large scale nature and the high dynamism of the targeted systems. Nevertheless, the research on adaptive stabilization show the importance of building systems that self-adapt to observed and current behaviors. Correction-on-use and correction-on-change presented in the following sub-sections provide this self-adaption without the need for separate sub-routines to be run periodically.

7.3 Correction-on-Use

Periodic stabilization is expensive and induces unnecessary bandwidth consumption. To overcome this problem an alternative approach, *correction-on-use*, is proposed in [2]. The idea here is to take advantage of the use of the overlay network in order to let it self-organizes in face of changes.

When a peer n joins a DKS network, it receives approximate routing information, that is not necessarily accurate. This routing information becomes accurate over time when the system is used. To achieve this convergence, two ideas are used: (i) whenever a peer receives a message from another peer, the receiving peer adapts itself to account the presence of the sender. (ii) whenever a peer n sends message to another peer n', peer n embeds some information about its current "local view" of the network. This local view is accurate thanks to the embedding of the $k-$ary trees. The receiving peer n' can then precisely determine whether the sender n had a correct view at the sending time. If not, a badpointer notification is sent back to peer n. The notification message carries the identifier of a candidate peer for correction. Upon receipt of such a notification, the sender peer n corrects itself.

If the ratio of the use (traffic injected into the system) over the dynamism of the system is high enough, the overlay network converges to a legitimate configuration. The main advantage of the correction-on-use is that it completely eliminates unnecessary bandwidth consumption. Each peer pays for what it needs. However, if the ratio of the traffic injected into the system over the dynamism is not sufficiently high, the convergence of the overlay network to legitimate configuration is slowed down.

7.4 Correction-on-Change

The correction-on-use technique presented in the previous section is useful as it eliminates unnecessary bandwidth consumption under the assumption that the traffic injected into the system is high enough for corrections to take place.

However, in some usage scenarios, it cannot be guaranteed that the traffic is high enough. For those scenarios we combine correction-on-use with a technique we call *correction-on-change* [12]. In the correction-on-change technique, whenever a change is detected, *all* peers that *depend* on the peer where the change occurred are corrected. We call the set of peers that depend on a given peer p, the set of dependent peers of p. A peer p is dependent on another peer p' if and only if peer p' should be in the routing table of peer p in legitimate configurations.

Whenever a peer p joins, leaves, or fails, the dependent peers of p are notified such that they can adjust their routing information accordingly. To implement this notification in an efficient manner, we use a restricted version of the correcting broadcasting algorithms that are being developed by our team. Hence, all dependent peers will be updated in parallel.

One consequence of correction-on-change is that it does not make leaves equivalent to failures. Second, whenever a failure is detected, all dependent peers are eagerly notified. Consequently, the system becomes more robust in the face of high dynamism. At the same time, in accordance with correction-on-use, no unnecessary bandwidth will be consumed during steady periods when the dynamism in the system is low. Full evaluation of this technique and its combination with correction-on-use is to be reported in another paper.

8 Conclusion

In this paper, we presented a framework for understanding, analyzing and designing structured peer-to-peer overlay networks. The proposed framework builds upon the principle of embedding $k-$ary trees into the virtual identifier space. Using the proposed framework, several variants of structured peer-to-peer overlay networks can be derived. The designer only need to decide which division of the space to use and the rule for selecting responsible peers. Many existing structured peer-to-peer overlay networks, such as [2, 26, 23, 24, 2, 1, 19, 16, 15], fit the presented framework.

Interestingly, from our framework of embedding $k-$ary trees, we can derive structured overlay networks of constant degree. In this paper, we briefly shown one way to achieve this. We report further algorithmic and simulations-based studies of constant degree structured peer-to-peer overlay networks in a full paper.

Given the embedding of k-ary trees, the proofs of correctness regarding logarithmic lookup length becomes trivial. Also, the understanding of the routing process is simplified due to the nature of interval routing that the embedded trees allow. Indeed, we show by this framework that routing in structured peer-to-peer overlay networks is essentially an interval/compact routing process.

The embedding of $k-$ary trees also has an impact on the design of high level services. For example, we have developed optimal one-to-many communication primitives [10, 13] based on the embedding of virtual $k-$ary trees on the virtual space. The derived algorithms inherit correctness properties as well as self-organization of the underlying substrate, which is a great advantage.

With the embedding of $k-$ary trees, a number of effective techniques for maintaining overlay networks are made possible. The correction-on-use is one such techniques. To increase robustness while keeping the maintenance cost low, we combine correction-on-use with correction-on-change. Correction-on-change eagerly corrects outdated routing pointers upon each change in the network. As a result, of this combination of correction-on-use and correction-on-change, unnecessary bandwidth consumption is avoided. In addition to these techniques, we are also investigating suitable adaptive maintenance techniques that combine effectively with correction-on-use and correction-on-change.

In this framework, we have been assuming that participating peers are homogeneous. In practice, this is not usually the case. We are therefore exploring techniques that will make use of our framework while integrating heterogeneity of peers.

References

1. Karl Aberer, *P-Grid: A self-organizing access structure for P2P information systems*, Lecture Notes in Computer Science **2172** (2001), 179–194.
2. L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi, *DKS(N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications*, The 3rd International workshop CCGRID2003 (Tokyo, Japan), May 2003.
3. L. O. Alima, A. Ghodsi, P. Brand, and S. Haridi, *Multicast in DKS(N, k, f) Overlay Networks*, 7th International Conference on Principles of Distributed Systems (OPODIS) (La Martinique, France), December 2003.
4. ———, *Multicast in DKS(N, k, f) Overlay Networks*, The 7th International Conference on Principles of Distributed Systems (OPODIS'2003) (Berlin), Springer-Verlag, 2004.
5. M. Amnefelt and J. Svenningsson, *Keso - a scalable, reliable and secure read/write peer-to-peer file system*, 2004.
6. M. Castro, P. Druschel, A-M. Kermarrec, and A. Rowstron, *SCRIBE: A large-scale and decentralised application-level multicast infrastructure*, IEEE Journal on Selected Areas in Communications (JSAC) (Special issue on Network Support for Multicast Communications (2002).

7. Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica, *Wide-area cooperative storage with CFS*, Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP '01) (Chateau Lake Louise, Banff, Canada), October 2001.
8. Frank Dabek, Ben Zao, Peter Druschel, John Kubiatowicz, and Ion Stoica, *Towards a common api for structured peer-to-peer overlays*, Proceedings of the Second International Workshop on Peer-to-Peer Systems, IPTPS, 2003.
9. S. El-Ansary, L. O. Alima, P. Brand, and S. Haridi, *A Framework for Peer-To-Peer Lookup Services Based on k-ary Search*, Tech. Report TR-2002-06, SICS, May 2002.
10. ———, *Efficient Broadcast in Structured P2P Netwoks*, 2nd International Workshop on Peer-to-Peer Systems (IPTPS '03), February 2003.
11. FreeNet, *http://freenet.sourceforge.net*, 2003.
12. A. Ghodsi, L. O. Alima, P. Brand, and S. Haridi, *Increasing Robustness while Minimizing Bandwidth Consumption in Structured Overlay Networks*, Tech. Report ISRN KTH/IMIT/LECS/R-03/07–SE, Kista Sweden, 2003.
13. A. Ghodsi, L. O. Alima, S. El-Ansary, P. Brand, and S. Haridi, *Self-correcting broadcast in distributed hash tables*, Parallel and Distributed Computing and Systems (PDCS'2003) (Calgary), ACTA Press, 2003.
14. Gnutella, *http://www.gnutella.com*, 2003.
15. N. J. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman, *SkipNet: A Scalable Overlay Network with Practical Locality Properties*, Fourth USENIX Symposium on Internet Technologies and Systems (USITS) (Seattle, USA), March 2003.
16. F. Kaashoek and D. R. Karger, *Koorde: A simple degree-optimal distributed hash table*, Proceedings of the Second International Workshop on Peer-to-Peer Systems, IPTPS, 2003.
17. John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao, *Oceanstore: An architecture for global-scale persistent storage*, Proceedings of the Ninth international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS 2000), November 2000.
18. D. Malki, M. Naor, and D. Ratajczak, *Viceroy: A scalable and dynamic emulation of the butterfly*, Proceedings of the 21st ACM Symposium on Principles of Distributed Computing, 2002.
19. Petar Maymounkov and David Mazires, *Kademlia: A Peer-to-peer Information System Based on the XOR Metric*, The 1st Interational Workshop on Peer-to-Peer Systems (IPTPS'02), 2002.
20. Ratul Mahajan Miguel, *Controlling the cost of reliability in peer-to-peer overlays*, Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS 03), 2003.
21. M. Naor and U. Wieder, *A simple fault tolerant distributed hash table*, Proceedings of the Second International Workshop on Peer-to-Peer Systems, IPTPS, 2003.
22. B. Clifford Neumann, *Scale in distributed systems*, pp. 463–489, IEEE Computer Society, Los Alamitos, CA, 1994.
23. Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker, *A Scalable Content Addressable Network*, Tech. Report TR-00-010, Berkeley, CA, 2000.

24. A. Rowstron and P. Druschel, *Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems*, Lecture Notes in Computer Science **2218** (2001).
25. I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan, *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*, ACM SIGCOMM 2001 (San Deigo, CA), August 2001, pp. 149–160.
26. ———, *Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications*, Tech. Report TR-819, MIT, January 2002, http://www.pdos.lcs.mit.edu/chord/papers/chord-tn.ps.
27. G. Tel, *Introduction to distributed algorithms*, Cambridge University Press, 1994, ISBN 0 521 47069 2.
28. Ben Y. Zhao, John D. Kubiatowicz, and Anthony D. Joseph., *Tapestry: An Infrastructure for Fault-tolerant Wide-area Location and Routing*, U. C. Berkeley Technical Report UCB//CSD-01-1141, April 2000.

Verifying a Structured Peer-to-Peer Overlay Network: The Static Case*

Johannes Borgström[1], Uwe Nestmann[1],
Luc Onana[2,3], and Dilian Gurov[2,3]

[1] School of Computer and Communication Sciences, EPFL, Switzerland
[2] Department of Microelectronics and Information Technology,
KTH, Sweden
[3] SICS, Sweden

Abstract. Structured peer-to-peer overlay networks are a class of algorithms that provide efficient message routing for distributed applications using a sparsely connected communication network. In this paper, we formally verify a typical application running on a fixed set of nodes. This work is the foundation for studies of a more dynamic system.

We identify a value and expression language for a value-passing CCS that allows us to formally model a distributed hash table implemented over a static DKS overlay network. We then provide a specification of the lookup operation in the same language, allowing us to formally verify the correctness of the system in terms of observational equivalence between implementation and specification. For the proof, we employ an abstract notation for reachable states that allows us to work conveniently up to structural congruence, thus drastically reducing the number and shape of states to consider. The structure and techniques of the correctness proof are reusable for other overlay networks.

1 Introduction

In recent years, decentralised structured peer-to-peer (p2p) overlay networks [OEBH03, SMK+01, RD01, RFH+01] have emerged as a suitable infrastructure for scalable and robust Internet applications. However, to our knowledge, no such system has been formally verified.

One commonly studied application is a *distributed hash table* (DHT), which usually supports at least two operations: the insertion of a (key,value)-pair and the lookup of the value associated to a given key. For a large p2p system (millions of nodes), careful design is needed to ensure the correctness and efficiency of these operations, both in the number of messages sent and the expected delay, counted in message hops. Moreover, the sheer number of nodes requires a sparse (but adaptable) overlay network.

* Supported by the EU-project IST-2001-33234 PEPITO (http://www.sics.se/pepito), part of the FET-initiative *Global Computing*.

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 250–265, 2004.

The DKS System

In the context of the EU-project PEPITO, one of the authors is developing a decentralised structured peer-to-peer overlay network called DKS (named after the routing principle *distributed k-ary search*), of which the preliminary design can be found in [OEBH03]. DKS builds upon the idea of *relative division* [OGEA$^+$03] of the virtual space, which makes each participant the root of a virtual spanning tree of logarithmic depth in the number of nodes.

In addition to *key-based routing* to a single node, which allows implementation of the DHT interface mentioned above, the DKS system also offers key-based routing either to all nodes in the system or to the members of a multicast group. The basic technique used for maintaining the overlay network, *correction-on-use*, significantly reduces the bandwidth consumption compared to its earlier relatives such as Chord [SMK$^+$01], Pastry [RD01] and Can [RFH$^+$01].

Given these features, we consider the DKS system as a good candidate infrastructure for building novel large-scale and robust Internet applications in which participating nodes share computing resources as equals.

Verification Approach

In this paper, we present the first results of our ongoing efforts to formally verify DHT algorithms. We initially focus on *static* versions of the DKS system: (1) they comprise a fixed number of participating nodes; (2) each node has access to perfectly accurate routing information. As a matter of fact, already for static systems formal arguments about their correctness turn out to be non-trivial.

We consider the correctness of the *lookup* operation, because this operation is the most important one of a hash table: under all circumstances, the data stored in a hash table must be properly returned when asked for. (The insert operation is simpler to verify: the routing is the same as for lookup, but no reply to the client is required.)

We analyse the correctness of lookup by following a tradition in *process algebra*, according to which a reactive system may be formulated in two ways. Assuming a suitably expressive process calculus at our disposal, we may on the one hand *specify* the DHT as a very simple purely sequential monolithic process, where every (lookup) request immediately triggers the proper answer by the system. On the other hand, we may *implement* the DHT as a composition of concurrent processes—one process per node—where client requests trigger internal messages that are routed between the nodes according to the DKS algorithm. The process algebra tradition says that if we cannot distinguish—with respect to some sensible notion of equivalence—between the specification and the implementation regarded as black-boxes from a client's point of view, then the implementation is correct with respect to the specification.

Contributions

While the verification follows the general approach mentioned above, we find the following individual contributions worth mentioning explicitly.

1. We identify an appropriate expression and value language to describe the virtual identifier space, routing tables, and operations on them.
2. We fix an asynchronous value-passing process calculus orthogonal to this value language and give an operational semantics for it.
3. We model both a specification and an implementation of a static DKS-based DHT in this setting.
4. We formally prove their equivalence using weak bisimulation. In detail:
   - We formalise transition graphs up to structural congruence.
   - We develop a suitable proof technique for weak bisimulation.
   - We design an abstract high-level notation for states that allows us to succinctly capture the transition graphs of both the implementation and the specification up to structural congruence.
   - We establish functions that concisely relate the various states of specification and implementation.
   - We show normalisation of all reachable states of the implementation in order to establish the sought bisimulation.

The proofs are found in the long version of the paper, which is accessible through http://lamp.epfl.ch/pepito.

Paper Overview

In Section 2 we provide a brief description of the DKS lookup algorithm, and identify the data types and functions used therein. In Section 3, we introduce a process calculus that is suitable for the description of DHT algorithms. More precisely, we may both specify and implement a DKS-based DHT in this calculus, as we do in Section 4. Finally, in Section 5 we formally prove that DKS allows to correctly implement the lookup function of DHTs by establishing a bisimulation containing the given specification and implementation.

Related Work

To our knowledge, no peer-to-peer overlay network has yet been formally verified. That said, papers describing such algorithms often include pseudo-formal reasoning to support correctness and performance claims.

Previous work in using process calculi to verify non-trivial distributed algorithms includes, e.g., the two-phase commit protocol [BH00] and a fault-tolerant consensus protocol [NFM03]. However, in these algorithms, in contrast to overlay networks, each process communicates directly with every other process.

Other formal approaches, for instance I/O-automata [LT98] have been used to verify traditional (i.e., logically fully connected) distributed systems; we are not aware, though, of any p2p-examples.

Future Work

Peer-to-peer algorithms in general are likely to operate in environments with high dynamism, i.e., frequent joins, departures and failures of participating nodes.

This case gives us increased complexity in three different dimensions: a more expressive model, bigger algorithms and more complex invariants.

To cope with dynamism, structured peer-to-peer overlay networks are designed to be stabilising. That is, if ever the dynamism within the system ceases, the system should converge to a *legitimate* configuration. Proving, formally, that such a property is satisfied by a given system is a challenge that we are currently addressing in our effort to verify peer-to-peer algorithms.

The work present in this paper is a necessary foundation for the more challenging task of formal verification of the DKS system in a dynamic environment.

Conclusions

The use of process calculi lets us verify executable formal models of protocols, syntactically close to their descriptions in pseudo-code. We demonstrate this by verifying the DKS lookup algorithm. Our choice to work with a reasonably standard process calculus, rather than the pseudo-code that these algorithms are expressed in, made it only slightly harder to ensure that the model corresponded to the actual algorithm but let us use well-known proof techniques, reducing the total amount of work.

Other overlay networks, like the above-mentioned relatives of DKS, would require changes to the expression language of the calculus as well as the details of the correspondence proof; however, we strongly conjecture that the structure of the proof would remain the same.

2 DKS

In this section we briefly describe the DKS system, focusing on the lookup algorithm. More information about the DKS system can be found for instance in [OEBH03, OGEA$^+$03].

For the design of the DKS system, we model a distributed system as a set of *processes* linked together through a *communication network*. Processes communicate by message passing and a process reacts upon receipt of a message; i.e., this is an event-driven model. The communication network is assumed to be (*i*) *connected*, each process can send a message directly to any other process in the system; (*ii*) *asynchronous*, the time taken by the communication network to forward a message to its destination can be arbitrarily long; (*iii*) *reliable*, messages are neither lost nor duplicated.

2.1 The Virtual Identifier Space

For DKS, as for other structured peer-to-peer overlay networks [SMK$^+$01, RD01], participating nodes are uniquely identified by identifiers from a set called *identifier space*. As in Chord and Pastry, the identifier space for DKS is a ring of size N that we identify with $\mathbb{Z}_N$, where we write $\mathbb{Z}_n$ for $\{0, 1, \cdots, n-1\}$. To model the ring structure, we let $\oplus$ and $\ominus$ be addition and subtraction modulo N, with the convention that the results of modular arithmetic are always non-negative and strictly less than the modulus. For simplicity, it is assumed that $N = k^d$ for $k > 1$, $d > 1$, where k will be the *branching factor* of the search tree. We work with a static system, with a fixed set of participating nodes $\mathcal{I} \subseteq \mathbb{Z}_N$ with $|\mathcal{I}| > 1$.

2.2 Assignment of Key-Value Pairs to Nodes

As part of the specification of a DHT, we assume that data items to be stored into and retrieved from the system are pairs $(\mathit{key}, \mathit{val}) \in \mathbb{N} \times \mathbb{N}$ where the keys are assumed to be unique. We model the data items currently in the system as a partial function $\mathrm{data} : \mathbb{N} \rightharpoonup \mathbb{N}$. Using some arbitrary hashing function, $\mathrm{H} : \mathbb{N} \to \mathbb{Z}_N$, the *key* of a data item is hashed to obtain a *key identifier* $\mathrm{H}(\mathit{key})$ for the pair $(\mathit{key}, \mathit{val})$.

In DKS (as well as in Chord), a data item $(\mathit{key}, \mathit{val})$ is stored at the first node succeeding $\mathrm{H}(\mathit{key})$. That node is called the *successor* of $\mathrm{H}(\mathit{key})$, and is defined as $\mathrm{suc}(i) \in \{j \in \mathcal{I} \mid j \ominus i = \min\{h \ominus i \mid h \in \mathcal{I}\}\}$. Note that $\mathrm{suc}(\cdot)$ is well-defined since $h \ominus i = j \ominus i$ iff $h \ominus j = 0$. Dually, the (strict) predecessor of a node $i \in \mathcal{I}$ is $\mathrm{pre}(i) \in \{j \in \mathcal{I} \mid j \ominus i = \max\{h \ominus i \mid h \in \mathcal{I}\}\}$. Local lookup at node n is a partial function $\mathrm{data}_n(j) := \mathrm{data}(j)$ if $\mathrm{suc}(j) = n$, i.e., returning the value $\mathrm{data}(j)$ associated to a key j only on the node n responsible for the item $(\mathit{key}, \mathit{val})$.

2.3 Routing Tables

The DKS system is built in a way that allows any node to reach any other node in *at most* $\log_k(N)$ hops under normal system operation. To achieve this, the principle of *relative division* of the space [OGEA+03] is used to embed, at each point of the identifier space, a complete virtual k-ary tree of height $d = \log_k(N)$. We let $\mathcal{L} := \{1, 2, \cdots, d\}$ be the levels of this tree, where 1 is the top level (the root). At a level $l \in \mathcal{L}$, a node n has a *view* V^l of the identifier space. The view V^l consists of k equal parts, denoted I_i^l, $0 \leq i \leq k-1$, and defined below level by level.

At level 1: $V^1 = I_0^1 \uplus I_1^1 \uplus I_2^1 \uplus \cdots \uplus I_{k-1}^1$, where $I_0^1 = [x_0^1, x_1^1)$, $I_1^1 = [x_1^1, x_2^1)$, $\cdots$, $I_{k-1}^1 = [x_{k-1}^1, x_0^1)$, $x_i^1 = n \oplus i\frac{N}{k}$, for $0 \leq i \leq k-1$.

At level $2 \leq l \leq d$: $V^l = I_0^l \uplus I_1^l \uplus I_2^l \uplus \cdots \uplus I_{k-1}^l$, where $I_0^l = [x_0^l, x_1^l)$, $I_1^l = [x_1^l, x_2^l)$, $\cdots$, $I_{k-1}^l = [x_{k-1}^l, x_1^{l-1})$, $x_i^l = n \oplus i\frac{N}{k^l}$, for $0 \leq i \leq k-1$.

To construct the routing table, denoted Rt_n, of an arbitrary node n of a DKS system we take for each level $l \in \mathcal{L}$ and each interval i at level l a pointer to the *successor* of x_i^l, as defined above.

Routing Table Example. As an example, consider an identifier space of size $N = 4^2$, i.e., $d = 2$ and $k = 4$. Assume that the nodes in the system are $\mathcal{I} := \{0, 2, 5, 10, 13\}$. In this case, using the principle described above for building routing table in DKS, we have that node 0 has the routing table in Figure 1.

Level	Interval	Responsible	Level	Interval	Responsible
1	$[0,4)$	0	2	$[0,1)$	0
	$[4,8)$	5		$[1,2)$	2
	$[8,12)$	10		$[2,3)$	2
	$[12,0)$	13		$[3,4)$	5

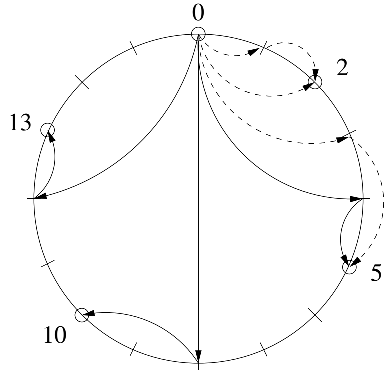


Fig. 1. Routing table for node 0

Formally, the routing tables of the nodes are partial functions

$$\mathrm{Rt}_n(j,l) := \mathrm{suc}\left(n \oplus \left(\frac{N}{k^l}\left\lfloor\frac{(j \ominus n)k^l}{N}\right\rfloor\right)\right) \text{ if } j \ominus n < k^{d+1-l} \text{ and } l \leq d,$$

where $\mathrm{Rt}_n(j,l)$ is the node responsible for the interval containing j on level l according to node n. We also define the lookup level for an identifier at a given node as $\mathrm{lvl}_n(j) := d - \lfloor \log_k(j \ominus n) \rfloor$, and let lookup in the routing table be $\mathrm{Rt}_n(j) := \mathrm{Rt}_n(j, \mathrm{lvl}_n(j))$, which is defined for all n, j.

2.4 Lookup in a Static DKS

The specification of lookup is common to all DHTs: A lookup for a key *key* at a node n should simply return the associated data value (if any) to the user on node n. Moreover, the system should always be available for new requests, and the responses may be returned in any order.

In DKS, the lookup can be done either iteratively, transitively or recursively. These are well-known strategies for resolving names in distributed systems [Gos91]. In this paper, we present a simplified version of the *recursive* algorithm of DKS.

Briefly and informally, the recursive lookup in the DKS system goes as follows. When a DKS node n receives a request for a key *key* from its user, u, node n checks if the virtual identifier associated to *key* is between $\mathrm{pre}(n)$ and n. If so, node n performs a local lookup and returns the value associated to *key* to the user. Otherwise, node n starts forwarding the request, such that it descends through the virtual k-ary tree associated with node n until the unique node z

such that H(*key*) is between pre(z) and z is reached. We call z the manager of *key*.

When the manager of *key* is reached, it does a local lookup to determine the value associated with *key*. This value is returned, back-tracing the path taken by the request. In order to do this, a stack is embedded in each internal request message, such that at each step of the forwarding process, the node n' handling the message pushes itself onto the stack. The manager z then starts a "forwarding" of internal response messages towards the origin of the request. Each such message carries the result of the lookup as well as the stack.

When a node n receives an internal response message, node n checks if the stack attached to the message is empty. If not, the head of the stack determines the next step in the "backwarding" of the message towards its origin. If the stack is empty, then n was the origin of the lookup. Then node n returns the result of the response to its user, u.

The back-tracing makes the response follow a "trusted path", to route around possible link failures, e.g., between the manager of the key and the originator of the lookup. The stack also provides some fault-tolerance: If the node at the head of the stack is no longer reachable, the nodes below can be used to return the message.

A formal model of this lookup algorithm can be found in Section 4, using the process calculus defined in Section 3.

3 Language

We use a variant of value-passing CCS [Mil89, Ing94] to implement the DKS system described above. To separate unrelated features and allow for a simple adaptation to the verification of other algorithms, we clearly distinguish three orthogonal aspects of the calculus.

Values and Expressions: The values $\mathcal{V}$ are integers, lists in nil $[\,]$ and cons $v_1 :: v_2$ format and the "undefined value" $\perp$. The expressions $\mathcal{E}$ contain some standard operations on values, plus common DHT functions and DKS-specific functions seen in Section 2.

We extend the domain and codomain of $F \in \{\mathrm{data}, \mathrm{lvl}_v, \mathrm{data}_v, \mathrm{Rt}_v \mid v \in \mathcal{I}\}$ to $\mathcal{V}$ by letting $F(v) := \perp$ for the values v on which F was previously undefined. We extend the domain of H to $\mathcal{V}$ by letting H take arbitrary values in $\mathbb{Z}_N$ for values not in $\mathbb{N}$. Expressions are evaluated using the function J·K : $\mathcal{E} \to \mathcal{V}$.

For boolean checks $\mathcal{B}$, we have the matching construct $e_1 = e_2$ and an interval check $e_1 \in (e_2, e_3\,]$ modulo N. Boolean checks are evaluated using the predicate $\mathbf{e}_{\mathrm{b}}(\cdot)$. Values and boolean checks are defined in Table 1, both J·K and $\mathbf{e}_{\mathrm{b}}(\cdot)$ are defined in Table 2. We do not use a typed value language, although the equivalence result obtained in Section 5.2 intuitively implies that the implementation is "as well-typed as" the specification.

We use tuples $\tilde{e}$ of expressions (and other terms), where $\tilde{e} := e_1, \ldots, e_{|\tilde{e}|}$ that may be empty, i.e., $|\tilde{e}| = 0$. To evaluate a tuple of expressions, we write $\llbracket\tilde{e}\rrbracket$ for the tuple of values $\llbracket e_1 \rrbracket, \ldots, \llbracket e_{|\tilde{e}|} \rrbracket$.

As a more compact representation of lists of values, we write $[u\tilde{v}]$ for $u :: [\tilde{v}]$, and also define $\mathrm{last}([v_1, v_2, \cdots, v_n]) := v_n$ if $n > 0$.

Parallel Language: We use a polyadic value-passing CCS, with asynchronous output and input-guarded choice. We assume that the set of names $a, b \in \mathcal{N}$ and the set of variables $x, y \in \mathcal{W}$ are disjoint and infinite. The syntax of the calculus can be found in Table 1.

As an abbreviation we write $\sum_{j\in\mathcal{J}} G_j$ for $\mathbf{0} + G_{j_0} + G_{j_1} + \cdots + G_{j_n}$ and $\prod_{j\in\mathcal{J}} P_j$ for $\mathbf{0} \,|\, P_{j_0} \,|\, P_{j_1} \,|\, \cdots \,|\, P_{j_n}$, where $\mathcal{J} = \{j_i \mid 0 \leq i \leq n\}$ ($\mathcal{J}$ may be $\emptyset$).

Control Flow Structures: We use the standard **if** ϕ **then** P **else** Q and a switch statement **case** e **of** $\{j \mapsto P_j \mid j \in S\}$ for a more compact representation of nested comparisons of the same value. In all **case** statements, we require $S \subset \mathcal{V}$ to be finite.

To gain a closer correspondence to the method-oriented style usually used when presenting distributed algorithms, we work with defining equations for process constants $\mathsf{A}\langle\tilde{e}\rangle$ rather than recursive definitions embedded in the process terms. If a process constant A does not take any parameters, we write A for both $\mathsf{A}\langle\rangle$ and $\mathsf{A}()$.

Table 1. Syntax

u, v	$::= 0, 1, 2, \cdots \quad \mid \quad [\,] \quad \mid \quad \perp \quad \mid \quad u :: u$	values $\mathcal{V}$
e	$::= u \quad \mid \quad x$	expressions $\mathcal{E}$
	$\mid \quad \mathsf{head}(e) \quad \mid \quad \mathsf{tail}(e) \quad \mid \quad e :: e$	(lists)
	$\mid \quad \mathsf{data}(e) \quad \mid \quad \mathsf{H}(e)$	(global)
	$\mid \quad \mathsf{lvl}_v(e) \quad \mid \quad \mathsf{data}_v(e) \quad \mid \quad \mathsf{Rt}_v(e)$	(local)
ϕ, ψ	$::= e = e$	boolean tests $\mathcal{B}$
	$\mid \quad e \in (e, e]$	(interval check)
G	$::= \mathbf{0}$	input-guarded sums $\mathcal{G}$
	$\mid \quad a(\tilde{x}).P$	(input prefix)
	$\mid \quad G + G$	(choice)
P, Q	$::= G$	processes $\mathcal{P}$
	$\mid \quad \overline{a}\langle\tilde{e}\rangle$	(asynchronous output)
	$\mid \quad P \,\vert\, P$	(parallel)
	$\mid \quad (P) \setminus a$	(restriction)
	$\mid \quad \mathsf{A}\langle\tilde{e}\rangle$	(process constant)
	$\mid \quad \textbf{if } \phi \textbf{ then } P \textbf{ else } P$	(if statement)
	$\mid \quad \textbf{case } e \textbf{ of } \{j \mapsto P_j \mid j \in S\}$	(case statement)

3.1 Semantics

The set of actions $\mathcal{A} \ni \mu$ is defined as $\mu ::= \tau \,|\, a\,\tilde{v} \,|\, \overline{a}\,\tilde{v}$. The channel of an action, $\mathrm{ch} : \mathcal{A} \to \mathcal{N} \cup \{\bot\}$, is defined as $\mathrm{ch}(\tau) := \bot$, $\mathrm{ch}(a\,\tilde{v}) := a$ and $\mathrm{ch}(\overline{a}\,\tilde{v}) := a$. The variables $\tilde{x}$ are bound in $a(\tilde{x}).P$. Substitution of the values $\tilde{v}$ for the variables $\tilde{x}$ in process P is written $P\,[{}^{v_1}/_{x_1}, \ldots, {}^{v_n}/_{x_n}]$ and performed recursively on the non-bound instances of $\tilde{x}$ in P. We use a standard labelled structural operational semantics with early input (see Table 2). To compute the values to be transmitted, instantiate process constants and evaluate **if** and **case** statements we use an auxiliary reduction relation $>$ (see Table 2).

Structural congruence is a standard notion of equivalence (cf. [MPW92]) that identifies process terms based on their syntactic structure. In a value-passing language, it often includes simplifications resulting from the evaluation of "top-level" expressions (cf. [AG99]). In our calculus, top-level evaluation is treated by the reduction relation $>$, which is contained in the structural congruence.

Definition 1 (Structural Congruence). Structural congruence $\equiv$ *is the least equivalence relation on* $\mathcal{P}$ *containing* $>$ *and satisfying commutative monoid laws for* $(\mathcal{P}, |, 0)$ *and* $(\mathcal{G}, +, 0)$ *and the following inference rules.*

$$\textsc{S-par}\ \frac{P_1 \equiv P_1'}{P_1 \,|\, P_2 \equiv P_1' \,|\, P_2} \qquad \textsc{S-sum}\ \frac{G_1 \equiv G_1'}{G_1 + G_2 \equiv G_1' + G_2} \qquad \textsc{S-res}\ \frac{P \equiv P'}{(P) \setminus a \equiv (P') \setminus a}$$

Depending on the actual structural congruence rules at hand it is well known, and can easily be shown, that structurally congruent processes give rise to the "same" transitions (leading to again structurally congruent processes) according to the operational semantics. Thus, transitions can be seen as a relation between congruence classes of processes. To simplify descriptions of the behaviour of processes, we define a related notion where we instead work with representatives for the congruence classes.

Definition 2 (Transition Graph Up to Structural Congruence). *A* transition graph up to structural congruence *is a labelled relation* $\Rrightarrow \subseteq \mathcal{Q}\times\mathcal{A}\times\mathcal{Q}$ *for* $\mathcal{Q} \subseteq \mathcal{P}$ *such that for all* $Q \in \mathcal{Q}$ *we have that*

– *If* $Q \xrightarrow{\mu} P'$, *there is* Q' *such that* $Q \overset{\mu}{\Rrightarrow} Q'$ *and* $P' \equiv Q'$.
– *If* $Q \overset{\mu}{\Rrightarrow} Q'$, *there is* P' *such that* $Q \xrightarrow{\mu} P'$ *and* $P' \equiv Q'$.

We say that $\Rrightarrow$ *is a transition graph up to* $\equiv$ *for* Q *if* $Q \in \mathcal{Q}$.

According to this definition, it is sufficient to include just one representative for the congruence class of a derivative; however, one may include several.

Weak bisimulation is a standard equivalence [Mil89] identifying processes with the same externally observable reactive behaviour, ignoring invisible internal activity. We define this process equivalence with respect to a general labelled transition system; this allows us to interpret the notion also on transition graphs up to $\equiv$.

Table 2. Semantics

Expression evaluation and boolean evaluation are defined as follows:

$$JeK := \begin{cases} v & \text{if } e = v \in \mathcal{V} \\ v_1 & \text{if } e = \mathsf{head}(e') \ \text{ and } Je'K = v_1 :: v_2 \\ v_2 & \text{if } e = \mathsf{tail}(e') \quad \text{ and } Je'K = v_1 :: v_2 \\ v_1 :: v_2 & \text{if } e = e_1 :: e_2 \quad \text{ and } Je_1K = v_1, Je_2K = v_2 \\ \mathrm{F}(Je'K) & \text{if } e = \mathsf{F}(e') \qquad \text{ and } \mathsf{F} \in \{\mathsf{data}, \mathsf{H}, \mathsf{lvl}_v, \mathsf{data}_v, \mathsf{Rt}_v \mid v \in \mathcal{I}\} \\ \bot & \text{if otherwise} \end{cases}$$

$$\begin{aligned} \mathbf{e}_{\mathrm{b}}(\, e_1 = e_2 \,) &\text{ is true iff } Je_1K = Je_2K \neq \bot \\ \mathbf{e}_{\mathrm{b}}(\, e_1 \in \, (e_2,\, e_3\,]\,) &\text{ is true iff } Je_iK = n_i \in \mathbb{N} \text{ for } i \in \{1,2,3\} \\ &\qquad \text{and } 0 < n_1 \ominus n_2 \leq n_3 \ominus n_2 \end{aligned}$$

The (top-level) *reduction relation* $>$ is the least relation on $\mathcal{P}$ satisfying:

1. $\overline{a}\langle\tilde{e}\rangle > \overline{a}\langle\tilde{v}\rangle$ if $J\tilde{e}K = \tilde{v}$.
2. $\mathsf{A}\langle\tilde{e}\rangle > P\left[{}^{v_1}\!/_{x_1}, \ldots, {}^{v_n}\!/_{x_n}\right]$ if $\mathsf{A}(\tilde{x}) \overset{\mathit{def}}{=} P$, $|\tilde{e}| = |\tilde{x}| = n$ and $J\tilde{e}K = \tilde{v}$.
3. **if** ϕ **then** P **else** $Q \ > P$ if $\mathbf{e}_{\mathrm{b}}(\,\phi\,)$.
4. **if** ϕ **then** P **else** $Q \ > Q$ if $\neg\, \mathbf{e}_{\mathrm{b}}(\,\phi\,)$.
5. **case** e **of** $\{j \mapsto P_j \mid j \in S\} > P_v$ if $JeK = v \in S$.

The structural operational semantics are given by the following inference rules, where the symmetric versions of Com-L, Par-L and Sum-L have been omitted.

$$(\text{IN})\ \frac{}{a(\tilde{x}).P \xrightarrow{a\,\tilde{v}} P\left[{}^{v_1}\!/_{x_1}, \ldots, {}^{v_n}\!/_{x_n}\right]}\ \text{ if } |\tilde{v}| = |\tilde{x}| \qquad (\text{OUT})\ \frac{}{\overline{a}\langle\tilde{v}\rangle \xrightarrow{\overline{a}\,\tilde{v}} \mathbf{0}}$$

$$(\text{COM-L})\ \frac{P \xrightarrow{a\,\tilde{v}} P' \qquad Q \xrightarrow{\overline{a}\,\tilde{v}} Q'}{P \,|\, Q \xrightarrow{\tau} P' \,|\, Q'} \qquad (\text{PAR-L})\ \frac{P \xrightarrow{\mu} P'}{P \,|\, Q \xrightarrow{\mu} P' \,|\, Q}$$

$$(\text{SUM-L})\ \frac{G_1 \xrightarrow{a\,\tilde{v}} P'}{G_1 + G_2 \xrightarrow{a\,\tilde{v}} P'} \qquad (\text{RES})\ \frac{P \xrightarrow{\mu} P'}{(P) \setminus a \xrightarrow{\mu} (P') \setminus a}\ \text{ if } a \neq \mathrm{ch}(\mu)$$

$$(\text{RED})\ \frac{P > Q \qquad Q \xrightarrow{\mu} Q'}{P \xrightarrow{\mu} Q'}$$

Definition 3 (Weak Bisimulation). *If* $\rightsquigarrow \subseteq \mathcal{P} \times \mathcal{A} \times \mathcal{P}$ *then a binary relation* $\mathcal{S} \subseteq \mathcal{P} \times \mathcal{P}$ *is a weak* $\rightsquigarrow$*-bisimulation if*
whenever $P \ \mathcal{S} \ Q$ *and* $P \overset{\mu}{\rightsquigarrow} P'$ *there exists* Q' *such that* $P' \ \mathcal{S} \ Q'$ *and*

- *if* $\mu = \tau$ *then* $Q \overset{\tau}{\rightsquigarrow}{}^{*} \ Q'$*;*
- *if* $\mu \neq \tau$ *then* $Q(\overset{\tau}{\rightsquigarrow}{}^{*}) \overset{\mu}{\rightsquigarrow} (\overset{\tau}{\rightsquigarrow}{}^{*})Q'$*,*

and conversely for the transitions of Q*.*

The notion usually deployed in process calculi is weak $\rightarrow$-bisimilarity: P is *weakly* $\rightarrow$*-bisimilar* to Q, written $P \approx Q$, if there is a weak $\rightarrow$-bisimulation $\mathcal{S}$ with $P\,\mathcal{S}\,Q$[1].

Next, we use the concept of $\Rrightarrow$-bisimilarity as simple proof technique: two processes are weakly $\rightarrow$-bisimilar if they are weakly $\Rrightarrow$-bisimilar.

Proposition 1. *If $\mathcal{S}$ is a weak $\Rrightarrow$-bisimulation, then $\equiv\mathcal{S}\equiv$ is a weak $\rightarrow$-bisimulation.*

4 Specification and Implementation

We now use the process calculus defined in Section 3 to specify and implement lookup in the DKS system.

Specification. In the specification process Spec, lookup requests and results are transmitted on indexed families of names $\mathit{request}_i, \mathit{response}_i \in \mathcal{N}$, where the index corresponds to the node the channel is connected to. The $\mathit{request}_i$ channels carry a single value: the key to be looked up. The $\mathit{response}_i$ channels carry the key and the associated data value.

$$\mathsf{Spec} \overset{\mathit{def}}{=} \sum_{i \in \mathcal{I}} \mathit{request}_i(\mathit{key}).(\overline{\mathit{response}_i}\langle \mathit{key}, \mathsf{data}(\mathit{key})\rangle \mid \mathsf{Spec}).$$

Implementation. The process implementing the DKS system, defined in Table 3, consists of a collection of nodes. A node Node_i is a purely reactive process that receives on the associated $\mathit{request}_i$, req_i and resp_i channels, and sends on $\mathit{response}_i$, req_j and resp_j for $j \in \mathrm{range}(\mathrm{Rt}_i(\cdot))$. The req_i channels carry three values: the key to be looked up, a stack specifying the return path for the result, and the current lookup level. The resp_i channels carry the key, the found value and the remaining return path.

Requests, i.e., messages on channels $\mathit{request}_i$ and req_i, are treated by the subroutine Req_i, which decides *whether* to respond to the message directly or to route it towards its destination. This decision is naturally based on whether it is itself responsible for the key searched for, as defined in Section 2; in this case, it responds with the value of a local lookup. Responses, i.e., messages on channels resp_i, are treated by the subroutine Resp_i, which decides *to whom* precisely to pass on the response; depending on the call stack, it either returns itself the result of a query to the application, or it passes on the response to the node from whom the request arrived earlier on.

The implementation of the static DKS system, Impl, is then simply the parallel composition of all nodes, with a top-level restriction on the channels that are not present in the DHT API. We use variables $\mathit{key}, \mathit{stack}, \mathit{value}, \mathit{level} \in \mathcal{W}$.

[1] The knowledgeable reader may note that although we find ourselves within a calculus with asynchronous message-passing, we use a standard synchronous bisimilarity, which is known to be strictly stronger than the notion of asynchronous bisimilarity. However, our correctness result holds even for this stronger version.

Table 3. DKS Implementation

$$\begin{aligned}\mathsf{Node}_i \overset{def}{=}\ & \mathit{request}_i(\mathit{key}).(\mathsf{Node}_i \mid \mathsf{Req}_i\langle \mathit{key}, [\,]\rangle)\\ +\ & \mathit{req}_i(\mathit{key}, \mathit{stack}, \mathit{level}).(\mathsf{Node}_i \mid \mathsf{Req}_i\langle \mathit{key}, \mathit{stack}\rangle)\\ +\ & \mathit{resp}_i(\mathit{key}, \mathit{value}, \mathit{stack}).(\mathsf{Node}_i \mid \mathsf{Resp}_i\langle \mathit{key}, \mathit{value}, \mathit{stack}\rangle)\end{aligned}$$

$$\begin{aligned}\mathsf{Req}_i(\mathit{key}, \mathit{stack}) \overset{def}{=}\ & \textbf{if}\quad \mathsf{H}(\mathit{key}) \in (\mathrm{pre}(i), i\,]\\ & \textbf{then}\ \mathsf{Resp}_i\langle \mathit{key}, \mathsf{data}_i(\mathit{key}), \mathit{stack}\rangle\\ & \textbf{else}\ \ \textbf{case}\ \mathsf{Rt}_i(\mathsf{H}(\mathit{key}))\\ & \qquad\quad \textbf{of}\quad \{j \mapsto \overline{\mathit{req}_j}\langle \mathit{key}, i :: \mathit{stack}, \mathsf{lvl}_i(\mathsf{H}(\mathit{key}))\rangle \mid j \in \mathcal{I}\}\end{aligned}$$

$$\begin{aligned}\mathsf{Resp}_i(\mathit{key}, \mathit{value}, \mathit{stack}) \overset{def}{=}\ & \textbf{if}\quad \mathit{stack} = [\,]\\ & \textbf{then}\ \overline{\mathit{response}_i}\langle \mathit{key}, \mathit{value}\rangle\\ & \textbf{else}\ \ \textbf{case}\ \mathsf{head}(\mathit{stack})\\ & \qquad\quad \textbf{of}\quad \{j \mapsto \overline{\mathit{resp}_j}\langle \mathit{key}, \mathit{value}, \mathsf{tail}(\mathit{stack})\rangle \mid j \in \mathcal{I}\}\end{aligned}$$

$$\mathsf{Impl} \overset{def}{=} \left(\prod_{i \in \mathcal{I}} \mathsf{Node}_i\right) \setminus \{\mathit{req}_i, \mathit{resp}_i \mid i \in \mathcal{I}\}$$

5 Correctness

Our correctness result is that the specification of lookup is weakly bisimilar to its (non-diverging) implementation in the DKS system. We show this by providing a uniform representation of the derivatives of the specification and the implementation, and their transition graphs up to $\equiv$, allowing us to directly exhibit the bisimulation.

5.1 State Space and Transition Graph

Since nodes are stateless (in the static setting), we only need to keep track of the messages currently in the system. For this we will use multisets, with the following notation: A *multiset* M over a set $\mathcal{M}$ is a function with type $\mathcal{M} \to \mathbb{N}$. By $\mathrm{spt}(M) := \{x \in \mathcal{M} \mid M(x) \neq 0\}$, we denote the *support* of M. We write 0 for any multiset with empty support. We can add and remove items by $S + a := \{a \mapsto S(a){+}1\} \cup \{x \mapsto S(x) \mid x \in \mathrm{dom}(S) \setminus \{a\}\}$ when $a \in \mathrm{dom}(S)$ and $S - a := \{a \mapsto S(a){-}1\} \cup \{x \mapsto S(x) \mid x \in \mathrm{dom}(S) \setminus \{a\}\}$, where $S - a$ is defined only when $a \in \mathrm{spt}(S)$. More generally, we define the sum of two multisets with the same domain as $S + T := \{x \mapsto S(x){+}T(x) \mid x \in \mathrm{dom}(S)\}$.

Specification. The states of the lookup specification are uniquely determined by the undelivered responses. To describe this state space, we define families of process constants $\mathsf{Responses}_\alpha$ and Spec_α, where α ranges over multisets with domain $\mathcal{I} \times \mathcal{V}$ and finite support. We write $t < n$ for $t \in \mathbb{Z}_n$.

$$\mathsf{Responses}_\alpha \overset{def}{=} \prod_{(i,kv) \in \mathrm{spt}(\alpha)} \ \prod_{t<\alpha(i,kv)} \overline{\mathit{response}_i}\langle kv, \mathrm{data}(kv)\rangle$$

Let $\mathsf{Spec}_\alpha := \mathsf{Responses}_\alpha | \mathsf{Spec}$. Note that $\mathsf{Spec} \equiv \mathsf{Spec}_0$.

Lemma 1. Spec_0 *has the following transition graph up to* $\equiv$.

1. $\mathsf{Spec}_\alpha \xRightarrow{request_i\ kv} \mathsf{Spec}_{\alpha+(i,kv)}$ *if* $i \in \mathcal{I}$ *and* $kv \in \mathcal{V}$

2. $\mathsf{Spec}_\alpha \xRightarrow{\overline{response_i}\ kv, \mathrm{data}(kv)} \mathsf{Spec}_{\alpha-(i,kv)}$ *if* $(i, kv) \in \mathrm{spt}(\alpha)$

Implementation. For the implementation, we also have to keep track of $resp_i$ and req_i messages and the values that can be sent in them. Since the routing tables are correctly configured, there is a simple invariant on the parameters of the $\overline{req_i}\langle kv, L, m\rangle$ messages in the system: Such messages are either sent to the node responsible for kv, or to the node responsible for the interval containing $\mathrm{H}(kv)$ on level m as discussed in Section 2. To capture this invariant we let $list[\mathcal{I}] := \{[i_1, i_2, \cdots, i_n] \mid i_j \in \mathcal{I} \wedge n \in \mathbb{N}\}$, and define $\mathcal{R} \subset \mathcal{I} \times \mathcal{V} \times list[\mathcal{I}] \times \mathbb{Z}_{d+1}$ as

$$\mathcal{R} := \{(i, kv, L, m) \mid L \neq [\,] \wedge \\ (\,\mathrm{suc}(\mathrm{H}(kv)) = i \ \vee\ \mathbf{e}_\mathrm{b}(\,\mathrm{H}(kv) \in\ (i,\ i \oplus k^{d-m} \ominus 1\,]\,)\,)\}.$$

To model the internal messages in the DKS system, we define families of process constants Reqs_β and Resps_γ where α is as above, β ranges over multisets with domain $\mathcal{R}$ and finite support and γ ranges over multisets with domain $\mathcal{I} \times \mathcal{V} \times list[\mathcal{I}]$ and finite support as follows.

$$\mathsf{Reqs}_\beta \overset{def}{=} \prod_{(i,kv,L,m)\,\in\,\mathrm{spt}(\beta)} \ \prod_{t<\beta(i,kv,L,m)} \overline{req_i}\langle kv, L, m\rangle$$

$$\mathsf{Resps}_\gamma \overset{def}{=} \prod_{(i,kv,L)\,\in\,\mathrm{spt}(\gamma)} \ \prod_{t<\gamma(i,kv,L)} \overline{resp_i}\langle kv, \mathrm{data}(kv), L\rangle$$

The behaviour of the implementation is captured by the constants $\mathsf{Impl}_{\alpha,\beta,\gamma}$.

$$\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{def}{=} \left(\mathsf{Responses}_\alpha | \mathsf{Reqs}_\beta | \mathsf{Resps}_\gamma | \prod_{i\,\in\,\mathcal{I}} \mathsf{Node}_i\right) \setminus \{req_i, resp_i \mid i \in \mathcal{I}\}$$

Note that $\mathsf{Impl} \equiv \mathsf{Impl}_{0,0,0}$.

Lemma 2. $\mathsf{Impl}_{0,0,0}$ *has the following transition graph up to* $\equiv$.

1. $\mathsf{Impl}_{\alpha,\beta,\gamma} \xRightarrow{request_i\ kv} \mathsf{Impl}_{\alpha+(i,kv),\beta,\gamma}$
if $i \in \mathcal{I}$ *and* $\mathbf{e}_\mathrm{b}(\,\mathrm{H}(kv) \in\ (\mathrm{pre}(i),\ i\,]\,)$

2. $\mathsf{Impl}_{\alpha,\beta,\gamma} \xRightarrow{request_i\ kv} \mathsf{Impl}_{\alpha,\beta+(\mathrm{Rt}_i(\mathrm{H}(kv)),kv,[i],\mathrm{lvl}_i(\mathrm{H}(kv))),\gamma}$
if $i \in \mathcal{I}$ *and* $\neg\big(\ \mathbf{e}_\mathrm{b}(\,\mathrm{H}(kv) \in\ (\mathrm{pre}(i),\ i\,]\,)\big)$

3. $\mathsf{Impl}_{\alpha,\beta,\gamma} \xRightarrow{\overline{response_i}\ kv, \mathrm{data}(kv)} \mathsf{Impl}_{\alpha-(i,kv),\beta,\gamma}$
if $(i, kv) \in \mathrm{spt}(\alpha)$

4. $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\Rrightarrow} \mathsf{Impl}_{\alpha,\beta-(i,kv,h::L,m),\gamma+(h,kv,L)}$ *if* $(i, kv, h{::}L, m) \in \mathrm{spt}(\beta)$ *and* $\mathbf{e}_{\mathrm{b}}(\,\mathrm{H}(kv) \in\ (\mathrm{pre}(i),\ i\,]\,)$

5. $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\Rrightarrow} \mathsf{Impl}_{\alpha,\beta-(i,kv,L,m)+(\mathrm{Rt}_i(\mathrm{H}(kv)),kv,i::L,\mathrm{lvl}_i(\mathrm{H}(kv))),\gamma}$ *if* $(i, kv, L, m) \in \mathrm{spt}(\beta)$ *and* $\neg\big(\ \mathbf{e}_{\mathrm{b}}(\,\mathrm{H}(kv) \in\ (\mathrm{pre}(i),\ i\,]\,)\big)$

6. $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\Rrightarrow} \mathsf{Impl}_{\alpha,\beta,\gamma-(i,kv,h::L)+(h,kv,L)}$ *if* $(i, kv, h{::}L) \in \mathrm{spt}(\gamma)$

7. $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\Rrightarrow} \mathsf{Impl}_{\alpha+(i,kv),\beta,\gamma-(i,kv,[\,])}$ *if* $(i, kv, [\,]) \in \mathrm{spt}(\gamma)$

Having found the transition graphs of both the specification and the implementation up to structural congruence, we restrict ourselves to working with this transition system.

Definition 4. *Let* $\Rrightarrow$ *be the union of the relations in the statements of Lemma 1 and Lemma 2.*

Note that $\Rrightarrow$ is as transition graph up to structural congruence for both Spec_0 and $\mathsf{Impl}_{0,0,0}$.

5.2 Bisimulation

To relate the state spaces of the specification and the implementation, we define two partial functions $\mathrm{T}_{req} : \mathcal{R} \rightharpoonup (\mathcal{I} \times \mathbb{N})$ and $\mathrm{T}_{resp} : (\mathcal{I} \times \mathbb{N} \times \mathit{list}[\mathcal{I}]) \rightharpoonup (\mathcal{I} \times \mathbb{N})$ that map the parameters of *req* and *resp* messages, respectively, to those of the corresponding *response* messages as follows.

$$\mathrm{T}_{req}(i, kv, L, m) := (\mathrm{last}(L), kv)$$

$$\mathrm{T}_{resp}(i, kv, L) := \begin{cases} (\mathrm{last}(L), kv) & \text{if } L \neq [\,] \\ (i, kv) & \text{if } L = [\,] \end{cases}$$

Note that T_{req} is well-defined since $\mathrm{dom}(\mathrm{T}_{req}) = \mathcal{R}$, thus $L \neq [\,]$. We then lift these functions to the respective multisets of type β and γ.

$$\widehat{\mathrm{T}_{req}}(\beta) := \sum_{x \in \mathrm{spt}(\beta)} \{\, \mathrm{T}_{req}(x) \mapsto \beta(x)\,\}$$

$$\widehat{\mathrm{T}_{resp}}(\gamma) := \sum_{x \in \mathrm{spt}(\gamma)} \{\, \mathrm{T}_{resp}(x) \mapsto \gamma(x)\,\}$$

Here $\sum$ denotes indexed multiset summation. Finally, we abbreviate the accumulated expected visible responses due to pending requests by:

$$\widehat{\mathrm{T}}(\alpha, \beta, \gamma) := \alpha + \widehat{\mathrm{T}_{req}}(\beta) + \widehat{\mathrm{T}_{resp}}(\gamma)$$

The implementation has a well-defined behaviour on internal transitions, as the following two lemmas show. First, internal transitions does not change the equivalence classes under the equivalence induced by the $\widehat{\mathrm{T}}$-transformation.

Lemma 3. *If* $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\Rrightarrow} \mathsf{Impl}_{\alpha',\beta',\gamma'}$ *then* $\widehat{\mathrm{T}}(\alpha,\beta,\gamma) = \widehat{\mathrm{T}}(\alpha',\beta',\gamma')$*.*

Next, we investigate the behaviour of the implementation when performing *sequences* of internal transitions. We prove that $\mathsf{Impl}_{\alpha,\beta,\gamma}$ is strongly normalizing on τ-transitions: it may always reduce to $\mathsf{Impl}_{\widehat{\mathrm{T}}(\alpha,\beta,\gamma),0,0}$, and does so within a bounded number of τ-steps.

Lemma 4 (Normalization). *For all* $\mathsf{Impl}_{\alpha,\beta,\gamma}$*, we have that*

1. $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\not\Rrightarrow}$ *iff* $\mathrm{spt}(\beta) = \emptyset = \mathrm{spt}(\gamma)$.
2. *there exists* $n \in \mathbb{N}$ *such that whenever* $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\Rrightarrow}^{k} I$*, then* $k \leq n$*.*
3. *if* $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\Rrightarrow}^{*} I \overset{\tau}{\not\Rrightarrow}$*, then* $I = \mathsf{Impl}_{\widehat{\mathrm{T}}(\alpha,\beta,\gamma),0,0}$*.*
4. $\mathsf{Impl}_{\alpha,\beta,\gamma} \overset{\tau}{\Rrightarrow}^{*} \mathsf{Impl}_{\widehat{\mathrm{T}}(\alpha,\beta,\gamma),0,0}$*.*

We now proceed to the main result of the paper, stating that the reachable states of the specification and of the implementation—in each case captured by the respective transition systems up to structural congruence—are precisely related.

Theorem 2. *The Binary Relation.*

$$\{\, (\,\mathsf{Spec}_{\widehat{\mathrm{T}}(\alpha,\beta,\gamma)},\, \mathsf{Impl}_{\alpha,\beta,\gamma}\,) \;\mid\; \mathsf{Impl}_{\alpha,\beta,\gamma} \textit{ is defined}\,\}$$

is a weak $\Rrightarrow$*-bisimulation.*

Corollary 3. $\mathsf{Spec} \approx \mathsf{Impl}$*.*

Proof. Since $\mathsf{Spec} \equiv \mathsf{Spec}_0$ and $\mathsf{Impl} \equiv \mathsf{Impl}_{0,0,0}$, this follows from Theorem 2 and Proposition 1.

This equivalence does not by itself guarantee that the implementation is free from live-locks since weak bisimulation, although properly reflecting branching in transition systems, is not sensitive to the presence of infinite τ-sequences. However, their absence was proven in Lemma 4(*2*).

References

[AG99] M. Abadi and A. D. Gordon. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, 1999.

[BH00] M. Berger and K. Honda. The Two-Phase Commitment Protocol in an Extended pi-Calculus. In L. Aceto and B. Victor, eds, *Proceedings of EXPRESS '00*, volume 39.1 of *ENTCS*. Elsevier Science Publishers, 2000.

[Gos91] A. Goscinski. *Distributed Operating Systems, The Logical Design.* Addison-Wesley, 1991.

[Ing94] A. Ingólfsdóttir. *Semantic Models for Communicating Processes with Value-Passing.* PhD thesis, University of Sussex, 1994. Available as Technical Report 8/94.

[LT98] N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. Technical Report MIT/LCS/TM 373, MIT Press, Nov. 1998.

[Mil89] R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MPW92] R. Milner, J. Parrow and D. Walker. A Calculus of Mobile Processes, Part I/II. *Information and Computation*, 100:1–77, Sept. 1992.

[NFM03] U. Nestmann, R. Fuzzati and M. Merro. Modeling Consensus in a Process Calculus. In R. Amadio and D. Lugiez, eds, *Proceedings of CONCUR 2003*, volume 2761 of *LNCS*. Springer, Aug. 2003.

[OEBH03] L. Onana Alima, S. El-Ansary, P. Brand and S. Haridi. DKS (N, k, f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. In *CCGRID 2003*, pages 344–350, 2003.

[OGEA$^+$03] L. Onana Alima, A. Ghodsi, S. El-Ansary, P. Brand and S. Haridi. Design Principles for Structured Overlay Networks. Technical Report ISRN KTH/IMIT/LECS/R-03/01–SE, KTH, 2003.

[RD01] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.

[RFH$^+$01] S. Ratnasamy, P. Francis, M. Handley, R. Karp and S. Shenker. A Scalable Content Addressable Network. In *SIGCOMM 2001, San Diego, CA*. ACM, 2001.

[SMK$^+$01] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *SIGCOMM 2001, San Diego, CA*. ACM, 2001.

A Physics-Style Approach to Scalability of Distributed Systems*

Erik Aurell[1,2] and Sameh El-Ansary[2]

[1] Swedish Institute of Computer Science, Kista, Sweden
[2] Department of Physics, KTH-Royal Institute of Technology,
Stockholm, Sweden
{sameh, eaurell}@sics.se

Abstract. Is it possible to treat large scale distributed systems as physical systems? The importance of that question stems from the fact that the behavior of many P2P systems is very complex to analyze analytically, and simulation of scales of interest can be prohibitive. In Physics, however, one is accustomed to reasoning about large systems. The limit of very large systems may actually simplify the analysis. As a first example, we here analyze the effect of the density of populated nodes in an identifier space in a P2P system. We show that while the average path length is approximately given by a function of the number of populated nodes, there is a systematic effect which depends on the density. In other words, the dependence is both on the number of address nodes and the number of populated nodes, but only through their ratio. Interestingly, this effect is negative for finite densities, showing that an amount of randomness somewhat shortens average path length.

1 Introduction

In this paper, we propose a method for analyzing properties of large-scale distributed systems based on analogies with thermodynamics and statistical mechanics. That is, given a distributed system of size $\mathcal{N}$ exhibiting a property $\mathcal{P}$, we would like to know the behavior of $\mathcal{P}$ at system sizes $\mathcal{N}$ much greater than where direct simulation is feasible. Particularly we would like to know on the one hand if the description can in any way become simpler for large enough $\mathcal{N}$, or if there is a way to determine what is a "sufficiently large $\mathcal{N}$", so that no simulation of an even larger system is necessary, or is likely to reveal any new information.

Physics was the first science to encounter problems of this sort. The number $\mathcal{N}$ of molecules in a macroscopic body, say a liter of water, is about 10^{27}. On the microscopic level, all substances are made of atoms and molecules of the same basic type – different number of electrons, protons and neutrons – yet large lumps of the same kind of atoms or molecules make up substances with

* This work is funded by the Swedish funding agency VINNOVA, PPC project and the European PEPITO and EVERGROW projects.

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 266–272, 2004.

distinct qualities which we can percieve. Those can be density, pressure (of a gas at given density and temperature), viscocity (of a liquid), conductivity (of a metal), hardness (of a solid), how sound and light do or do not propagate, if the material is magnetic, and so on. Materials are important, indeed whole ages of human history have been named after the dominant material at the time [1].

The first level of analysis in a physical system of many components, is to try to separate *intensive* and *extensive* variables. Extensive variables are those that eventually become proportional to the size of the system, such as total energy. Intensive variables, such as density, temperature and pressure, on the other hand becomes independent of system size. A description in terms of intensive variables only is a great step forward, as it holds regardless of the size of the system, if sufficiently large.

Further steps in a physics-style analysis may include identifying phases, in each of which all intensive variables vary smoothly, and where the characteristics of the system remain the same. This methodology was carried over to satisfiability theory more than ten years ago. KSAT is the problem to determine if a conjunction of M clauses, each one a disjunction of K literals out of N variables can be satisfied. Both M and N are extensive variables, while $\alpha = M/N$, the average number of clauses per variable, is an intensive variable. For large N, instances of KSAT fall into either the SAT or the UNSAT phase depending on whether α is larger or smaller than a threshold $\alpha_c(K)$ [2, 3]. The order of the phase transition, a statistical mechanics concept roughly describing how abrupt the transition is, has been shown to be closely related to the average computational complexity of large instances of KSAT with given values of K and α [4, 5]. Recent advances include the introduction of techniques borrowed from the physics of disordered systems, leading to an important new class of algorithms, currently by far the best of large and hard SAT problems [6]. Without question, statistical mechanics have been proven to be very useful on very challenging problems in theoretical computer science, and it can be hoped that this will also be the case in the analysis and design of distributed systems.

2 The Physics-Style Analysis of Structured Overlays

We start our investigation by considering the area of Peer-To-Peer systems as an example of large-scale distributed systems. The investigation is done using the Chord system [7, 8] and familiarity of the reader to Chord concepts and terminology is assumed.

The work reported in this paper can then be summarized by the three following methodological steps:

Step 1: Determination of Intensive Variables. Let N be the size of the identifier space and P be the population, i.e. the number of nodes that are uniformly distributed in the identifier space. We define the density (ρ) to be the ratio $\frac{P}{N}$ with a maximum value of 1 for a fully populated system. We will here focus on the investigation of ρ as an intensive variable.

```
for $N \in \{2^7, 2^8, .., 2^{14}\}$ do
  for $P \in \{0.1 \times N, 0.2 \times N, .., 1.0 \times N\}$ do
    1. Generate CHORD($P$,$N$)
    2. Inject uniformly distributed $P^2$ lookups
    3. Record the average lookup length over
       the $P^2$ lookups, denoted $< L(P, N) >$ or
       equivalently $< L(\rho, N) >$
  end
end
```

Fig. 1. The procedure for investigating the density ($\rho = \frac{P}{N}$) as an intensive variable

Step 2: Looking for Characteristic Behavior. A key quantity of interest in a P2P system built of DHTs is the average path length. Here we report the dependency of the number of populated nodes, and the density in a series of simulations of Chord. We will show that while the main behaviour is $0.5 \log_2 P$, where P is the number of populated nodes, there is also a small residual term that depends on the density only.

Step 3: Ideas This Can Give to P2P Systems. It is a curious fact that the residual term alluded to above is negative. We call this curious, because if the P populated nodes are regularly spaced in the circular geometry of the address space of Chord, the average path length is exactly $0.5 \log_2 P$, in other words larger. Hence, we have as a result that randomization improves the performance of P2P system built on DHT, even in static situation, with no peers leaving or joining the system. We believe this may be of some conceptual importance, even if the effect is small.

Further ideas are taken up in the Discussion and Results section below.

3 The Simulation Setup

Let CHORD(P,N) be an optimal Chord graph, where all the fingers of all nodes are correctly assigned, our simulations follow the procedure illustrated in figure 1.

This procedure is repeated 10 times, with different random seeds, and the results are averaged.

The simulations were implemented using the Mozart distributed programming platform [9]. The total number of experiments performed was 800 that were scheduled on a cluster of 16 machines at SICS.

4 Results

Given the set of experiments performed as explained in figure 1, we report the results in a number of different ways.

First, as shown in figure 2, for a given set of P nodes, by placing them in identifier spaces of different sizes, the path length is affected. In fact, the path length decreases as the identifier space increases. The graph is based on a subset

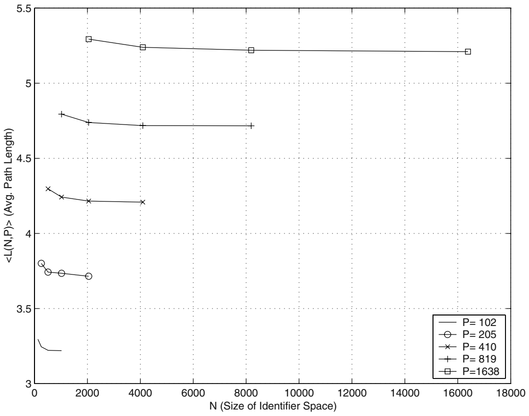


Fig. 2. The effect of different identifier space sizes on systems of the same population

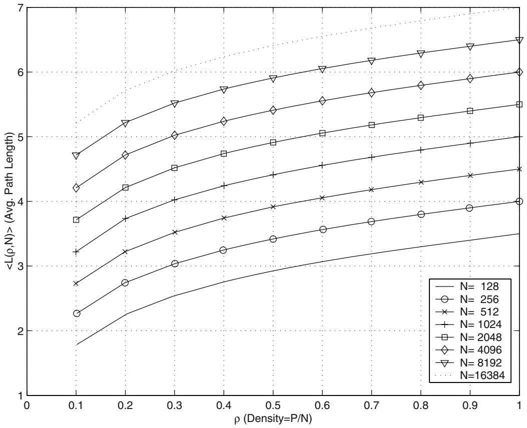


Fig. 3. The average lookup length as a function of ρ and N

of the data points where Ps are equal. This graph is mainly to show that the lookup length is not a function of the population alone.

Second, in figure 3 we show the behavior of the path length as a function of the density and the size of the identifier space. The curves are, to first approximation, vertically shifter by the same distance, while the values of N used are exponentially spaced. This means that the dependence on N alone (constant P) is logarithmic. Indeed, it was noted in the Chord papers that the average path length is $0.5 \times \log P$. However, we can see an additional observation by looking at the data collapse obtained in figure 4 by subtracting $< L(1, N) >$ from every respective curve $< L(\rho, N) >$ compared to $0.5 \log_2 \rho$. From the data collapse, we can clearly see that $< L(\rho, N) >= 0.5 \log_2 \rho + f(\rho)$ where the function f is a decreasing function. That is for any given number of nodes, the lookup length

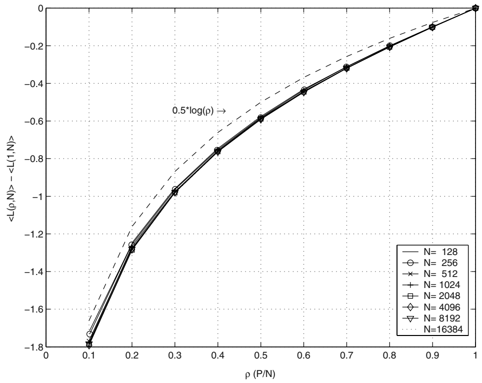


Fig. 4. Data collapse of the average lookup length as a function of ρ and N compared to $0.5 \log_2 \rho$

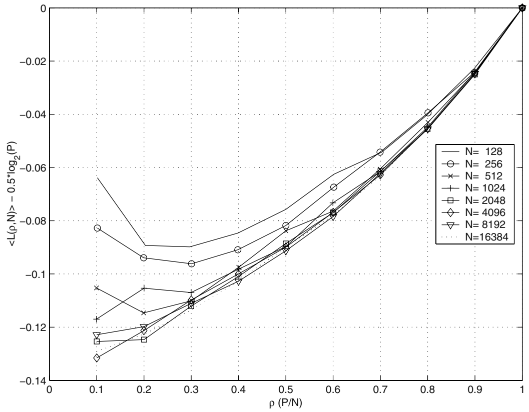


Fig. 5. Data collapse by subtracting the respective $0.5 \log_2 P$ from all data points

increases when they are placed in a smaller identifier space. Another way of observing this phenomena is shown in figure 5 where from each data point, the respective $0.5 \log_2 P$ is subtracted.

5 Discussion

A main direct result of this paper is that if one compares two Chord systems, both with P nodes, in address spaces of size N_1 and N_2, $N_1 < N_2$, the average path length in the (P, N_1) system is larger. This is somewhat counter-intuitive, since it suggests that having a larger space to look in speeds up the search (on average). Imagine that in both cases the address space is in fact of the same size

N_2, but in the first case only the N_1 regularly spaced nodes can be populated, with gaps of size N_2/N_1 between them.

The explanation is that the Chord routing table has $\log_2 N_2$ elements. If all N_2 addresses can be populated, all entries of the routing table can be used to list a hop that will bring you closer to your destination. In the situation with only N_1 addresses, there is rigidity in the placement of populated nodes (in the address space of size N_2), and all elements in the routing table are not used. Indeed, if written in the N_1 address space, the routing table has but $\log_2 N_1$ entries. Hence, the effect of having more keys to look for locally is slightly stronger than having to search in a larger spaces. In different language, given structured overlay network, it is appearently an advantage to sample it randomly, and not be confined to a subset of keys and possible nodes.

The overall motivation for this work is that physics-style analysis may prove useful in designing and analysing large P2P systems, and we end with a short summary of what we want to accomplish in this direction, and why.

First, it is a well-known fact in the community that simulations of different P2P systems of varying sizes are plentiful, but systematic methods to compare them are more rare. Statistical mechanics is the physical theory of what macroscopic properties that emerge for which microscopic descriptions, and how a large system approaches the infinitely large, or thermodynamic, limit. The ultimate goal when analysing a P2P system is indeed to to find out which desirable (or desirable) properties hold for a large system as a whole, when you have but specified the individual component. This is in fact a new statistical mechanics to be discovered. It is also of some practical importance to be able to say that a simulation is large enough, and that nothing essential can be gained by simulating an even larger system. Specific techniques for such tasks are e.g. finite-size scaling.

Second, dynamic properties should also be described by intensive variables. A possibility would be e.g. the average number of join or leave operations per populated node and time between running a stabilization algorithm. One example which has been studied by one of the authors is the ratio of join or leave operations to the number of look-ups generated in DKS [10], a system where updating outdated routing information is performed on the fly. Preliminary results on this system indicate the existence of at least two phases, one "good" (with path length proportional to $\log_2 P$), and one bad (with a very large path length, possibly proportional to P).

Third, it may be a general feature of many P2P systems, that they should preferentiably be operated in a "good" phase, but as close as possible to phase boundaries. This is because goodness usually costs, e.g. in stabilization. Techniques to estimate where phase boundaries lie, and to monitor if they are close in a dynamically changing environment, may hence give new ideas to control of P2P systems. Examples in this direction are spontaneous fluctuations, which grow strongly in size close to at least some types of phase transitions.

6 Conclusion

The main contribution of this position paper is the introduction of a methodology for analysing large-scale distributed systems using physical systems analysis techniques. The main goal of such an approach is to illustrate how the behaviors of large systems could be understood while eliminating the need to simulate large instances.

An example of the methodology is provided as a study of a property of the Chord system, namely the density of the population of nodes in an identifier space.

Acknowledgements

We thank Scott Kirkpatrick for discussions on the interface between P2P and statistical physics. We thank Seif Haridi, Per Brand, Luc Onana, Ali Ghodsi and the other members of the DSL laboratory at SICS for many discussions.

References

1. Chaikin, P.M., Lubensky, T.C.: Principles of condensed matter physics. Cambridge University Press (1995) ISBN: 0521794501.
2. Mitchell, D., Selman, B., Levesque, H.: Hard and easy distributions of sat problems. AAAI-92. Proceedings Tenth National Conference on Artificial Intelligence (1992) 873, 459–65
3. Kirkpatrick, S., Selman, B.: Critical behaviour in the satisfiability of random boolean expressions. Science **264** (1994) 1297–1301
4. Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., Troyansky, L.: Determining computational complexity from characteristic phase transitions. Nature **400** (1999) 133–137
5. Monasson, R., Zecchina, R., Kirkpatrick, S., Selman, B., Troyansky, L.: 2+p-sat: Relation of typical-case complexity to the nature of the phase transition. Random Structures and Algorithms **3** (1999) 414
6. Mézard, M., Parisi, G., Zecchina, R.: Analytic and algorithmic solutions of random satisfiability problems. Science **297** (2002) 812–815
7. Stoica, I., Morris, R., Karger, D., Kaashoek, M., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In: ACM SIGCOMM 2001, San Deigo, CA (2001) 149–160
8. Stoica, I., Morris, R., Karger, D., Kaashoek, M., Balakrishnan, H.: Chord: A Scalable Peer-to-Peer Lookup Service For Internet Applications. Technical Report TR-819, MIT (2002) http://www.pdos.lcs.mit.edu/chord/papers/chord-tn.ps.
9. Mozart Consortium: The mozart homepage (2003) http://www.mozart-oz.org.
10. Alima, L.O., El-Ansary, S., Brand, P., Haridi, S.: DKS(N; k; f): A Family of Low Communication, Scalable and Fault-Tolerant Infrastructures for P2P Applications. In: The 3rd International Workshop On Global and Peer-To-Peer Computing on Large Scale Distributed Systems-CCGRID2003, Tokyo, Japan (2003) http://www.ccgrid.org/ccgrid2003.

BGP-Based Clustering for Scalable and Reliable Gossip Broadcast*

M. Brahami[1], P. Th. Eugster[2], R. Guerraoui[1], and S. B. Handurukande[1]

[1] Distributed Programming Laboratory,
Swiss Federal Institute of Technology in Lausanne (EPFL)
[2] Sun Microsystems, Switzerland

Abstract. This paper presents a locality-based dissemination graph algorithm for scalable reliable broadcast. Our algorithm scales in terms of both network and memory usage. Processes only have "local knowledge" about each other. They organize themselves dynamically (right from the bootstrapping phase), according to join, leave or crash events, to form a locality-based dissemination graph. Broadcast messages can be disseminated using these graphs in large networks like the Internet, without relying on any special infrastructure or intermediate brokers. Roughly speaking, a dissemination graph consists of "non-crossing" (independent) trees that provide multiple paths between processes for improved broadcast efficiency and reliability. Each tree is constructed using BGP routing information about process "locality". We convey the feasibility of the algorithm using both simulation and experimental results and describe an application of our algorithm for broadcasting information streams.

Keywords: System design, Peer-to-peer communications, Content distribution, Multicast, Service overlay networks, Fault-tolerance, Broadcast streams.

1 Introduction

Traditional reliable broadcast algorithms [1] guarantee a very high level of reliability, despite message losses and process crashes, but scale poorly. Broadcast schemes like IP multicast [2] and MBone [3] are not widely available in the Internet and need specially configured routers (multicast routers and mrouted processes). Further more, these schemes do not cope well with message losses and process crashes.

Application level broadcast algorithms [4–10] have been recently proposed as a viable alternative. The general idea is to make use of some intermediate overlay network over the actual physical network, in order to cluster processes and achieve more efficient information dissemination. Such application level broadcast algorithms have good scalable and reliable properties. They can also be deployed easily in a large scale setting. Many of these approaches exploit an abstract notion of "locality" in the clustering procedure in order to arrange processes according to network locality. Some approaches use centralized services [11, 12] to find "distance" between processes in the Internet.

* This work was sponsored by European Project PEPITO (IST 2001 33234).

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 273–290, 2004.

But, with a large number of processes, it becomes quickly infeasible to query a centralized service with all the existing process IDs to determine a "close" process for a given process. Round trip time (RTT) is also used to find "locality" information for clustering processes. But again this has the same draw back (the difficulty of finding RTT to many number of processes) as above with a large number of processes.

In this paper we present a locality-based broadcast algorithm. Our algorithm is scalable and provides a reasonable level of reliability. No specific infrastructure is needed (unlike for e.g., [3] and [13]). Processes have local knowledge about each other and self-organize (peer-to-peer based) in a dynamic content distribution scenario where processes join and leave in an ad-hoc fashion to form a dissemination graph. When a new process joins, it contacts one or more existing process/es to find a suitable place in the graph. In simple terms, this resembles a tree search to find a "locality" based suitable region. The notion of locality is based on BGP [14] routing information. Our scheme minimizes the usage of slow network links to reduce the delay incurred to deliver messages and reduce congestion in such links. It offers high level of reliability in a dynamic environment where processes leave the graph and crash. To achieve this, our graph building algorithm constructs "non-crossing" independent paths within the dissemination graph. We assume that there are few sources which broadcast with respect to the destinations. We show reliability properties using formal analysis and illustrate the feasibility of our approach through results obtained both with a prototype as well as with simulations.

In Section 2 we contrast our approach with related work. Our general broadcast architecture is presented in Section 3. The reliability of the scheme with independent trees is formally analysed in Section 4. A full algorithm which performs clustering and constructs dissemination graph with independent trees is described in Section 5. Section 6 presents simulation and prototype measurement results. Two possible applications of our general broadcast scheme is discussed in Section 7. Section 8 concludes this paper.

2 Related Work

BGP information is used in [15] to construct topological-aware Distributed Hash Tables (DHT). These DHTs are used to locate objects in an overlay network.

Application level broadcast have been widely described in the literature. We discuss here the efforts closest to ours.

In [7], several ways of arranging peers to form a hierarchy are mentioned, namely by selecting nodes either through 1) a random fashion 2) a round-robin fashion or 3) a smart-placement fashion (based on their network location). In a very large scale network, the network-oblivious schemes based on random and round-robin selections are obviously not adequate. A more efficient approach consists in selecting nodes using smart-placement. This is done in [7] through a centralized service. Though many such services are described (e.g., [11, 12]), to our knowledge, they are not available in the Internet to be utilized as such. Even if such a service was available, it would not be clear how it could be used in a large scale setting.

Narada [4] is a multicast scheme with self-organizing capabilities. In this scheme, every member maintains a list of all other members, as well as a list of routing cost to every other member for paths between them. Then a per-source tree is constructed using an algorithm similar to DVMRP [16]. The scheme is very heavy in terms of memory and hence does not scale. Indeed, and according to the authors, the scheme is targeted towards medium size groups (with hundreds of members).

Scattercast [6] is based on an infrastructure (a set of servers known as agents) which needs to be deployed a priori. The agents construct an overlay network using a method similar to [4]. Individual clients are attached to close proximity scattercast clusters. The method used for automatic location of such a scattercast cluster is not presented (it is identified as a subject of future work by the author).

YOID [5] is an application level broadcast scheme which uses a general concept of "locality". However, no concrete hint is given on how locality information is determined when interconnecting processes.

Gossip-based broadcast algorithms [8, 17, 13, 18, 19] can also be viewed as application level broadcast schemes. While providing probabilistic guarantees on delivering messages, they have very good scalability properties and high resilience against failures and message losses. For instance, the algorithm presented in [13] is targeted toward small-scale WANs and relies on "gossip servers" which need to be setup and configured for each LAN manually. Hierarchical gossip-based broadcast algorithms presented in [18, 19] promote the idea of grouping processes (members) according to their locality, but no concrete way to exploit this locality is presented.

SplitStream [20], a content distribution scheme built on top of a DHT uses multiple paths between the content source and the destination. The topological placement of nodes is not done globally but limited to a few number of nodes (the "leaf set" of the DHT).

We describe in this paper a deterministic approach to message forwarding that limits the network usage. BGP-based clustering scheme can be applied both in a deterministic as well as in a randomized gossiping context.

3 Dissemination Architecture

In our dissemination scheme, the processes self organize in a peer-to-peer fashion to form a dissemination graph. The basic idea of our scheme is conveyed in Figure 1. Depending on their available resources (and possibly user-defined criteria), a subset of processes can be used to forward the messages they receive to other processes. The processes which are not capable of forwarding events to others (e.g., due to resource constraints) act as "pure clients" and receive events forwarded either by the source or by some other processes.

We construct a graph for message dissemination in a peer-to-peer basis. There is only one source for a given graph. For the sake of presentation simplicity, yet without loss of generality, we only discuss one such graph. The graph consists of independent trees connecting the source and all receiving processes. These trees are constructed according to the network bandwidth between processes. Broadcast is achieved by forwarding messages along these trees.

Processes have parent-child relationships. A process P_1, which forwards the messages it receives to another process P_2, is called a *parent* of P_2 and P_2 is called a *child* of P_1. The number of possible children a parent can have is called the parent's *fanout*. The *fanout* of a process is chosen according to its capabilities in terms of computation and network resources. Our scheme aims at grouping processes which are "closer" in the Internet. This is achieved using BGP-based clustering as described below.

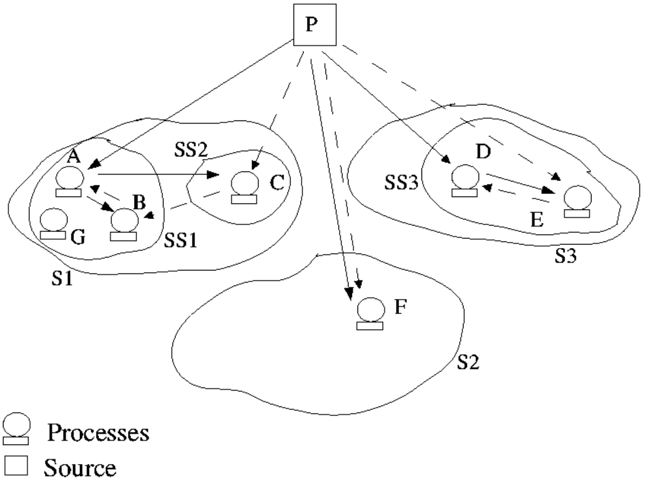


Fig. 1. Process-Assisted Broadcast

3.1 BGP-Based Clustering

Our clustering scheme relies on BGP [14] routing information. In particular, we use the notion of Autonomous System [14] (AS) to identify the segments of the Internet. Basically, an AS consists of a set of networks (a set of IP address blocks) which are managed by one administrative domain like an Internet Service Provider (ISP). Generally the networks inside a same AS have high bandwidth links connecting them. For example, the networks connecting universities in a country, large companies and organizations are typically in the same autonomous systems. In other terms, within such AS, individual networks and computers usually have high bandwidth links between them. Processes in a given autonomous system are arranged as neighbors in our dissemination graph.

It is possible for a process to find its AS using BGP information. This information is extracted using BGP routers or WHOIS servers [21, 22]. A service can be built on top of BGP routers to provide this information directly from the lower level network in a real time fashion. On the other hand, there are WHOIS servers [22, 23] which provide BGP information in an off-line fashion. This service can be mirrored locally, for example at the source, to have a low response time. Also the network address of a process (which has an IP) can be obtained using such BGP routers and WHOIS servers. The network address represents more fine-grained segmentation of the network.

In this paper, we only consider the AS and do not use the network address. Each new process can know its corresponding AS when the process first contacts the service when joining the group. If a fine-grained division is required, the network address of a given process can also be used in the clustering. In such a case, processes are also grouped according to the local networks they belong to inside a given AS.

We also group ASs according to their countries. This helps a process to aggregate information about other processes: as a result, first, the size of the memory necessary to keep local information about neighbors is minimized. Second, there is a high probability to find a "closer" parent to a given process within the same country than from a different country.

These techniques can be used to divide the network into clusters and sub-clusters. A group of ASs in a country is called *cluster* where as individual ASs are *sub-clusters* within the cluster. This approach to find the "locality" can be efficiently applied in a large scale setting like in the Internet.

The clustering of processes promotes good communication between neighbors and minimizes the amount of "local" information about other processes without keeping global knowledge. It also helps a new process to find a suitable parent within an acceptable time duration without consuming too much computing and network resources. In other words, though there are many thousands of processes, the new process is provided with adequate information to select a suitable parent.

3.2 Parent Selection

Our scheme provides heuristic information for any new process help it to find a suitable set of parents. The source and a set of existing processes provide this information. To ensure scalability, the source and processes keep a minimum amount of information for providing this heuristic information. The source keeps the information about the clusters and the knowledge about a limited set of processes for each of these clusters. When a new process contacts the source, the source first checks the cluster to which the new process belongs. Then the source, using its local knowledge, checks if there is any process already in this particular cluster. If the source finds one or more processes which are already in this cluster, then the new process is provided with the IDs of those existing processes (if there are no existing processes in the cluster, the new process will join as a child to the source itself). The new process contacts these processes which are already in the cluster. Unlike the source, the processes inside a cluster have more precise information about other processes in their respective clusters. That is, inside a cluster each process knows to which sub-cluster it belongs. Also, processes know a subset of the processes in their sub-cluster as well as in other sub-clusters. As a result, if the new process p_n is in AS_n, then p_n will be redirected to any existing processes in AS_n (details are in the Section 5). If there are no such existing processes in AS_n, p_n joins to some other existing processes.

The exact choice of a parent is made by the new process from the set of possible parents provided to the new process. In other terms, the new process measures the communication latency to each potential parent and also verifies the availability of an out-going link from each such parent. The parent is chosen such that it has an available out-going link with sufficiently low latency to the new process.

At this point, it should be noted that the trees can be re-arranged to make the scheme more efficient. For example, if a new process with a fanout greater than zero can not join because all the leaf processes of the tree are having a fanout of zero, then the tree will re-arrange by making the new process a new intermediate node in the tree.

Due to clustering, processes only keep a limited amount of information about other processes for the purpose of parent selection: this preserves scalability.

Example. The parent selection can be elaborated using a simple diagram as shown in Figure 1. In this simple example, the processes belong to three clusters S1, S2 and S3. The source P knows about processes A, C, F, D and E from clusters S1, S2 and S3 respectively. (For the purpose of having a higher reliability, the source can know more than one process from each cluster). In cluster S1, there are two sub-cluster SS1 and SS2. Processes A and B are located inside SS1 while C is in SS2. Process C, which is also known by P, knows that process B is a child (of C) and belongs to sub-cluster SS1.

Suppose a new process G from SS1 wants to receive events from P. G first contacts the source P. P observes that G belongs to S1. As P already knows that A and C are in S1, P asks G either to join A and C, or obtain more precise information from them. Once G contacts C (or A), C observes that G is in SS1. G will be informed about B by C. Depending on the available resources, G joins A and B as a child. In a situation where none of the existing processes in a cluster can be assigned as a parent to a new process (for example, due to resource constraints), the tree will be re-arranged to make the new process an internal node in the tree (assuming the new process can forward messages to other processes) and previous leaf processes will remain leaf processes, in the newly arranged tree. If all the existing processes are not able to forward messages, and the new process also can not forward messages, the new process will join the source. A similar scenario applies to process F in S2 since F is the only process in that cluster.

4 Independent Trees

At this point it should be clear that the processes in the lower level of the dissemination tree rely on the proper functioning of the higher level processes which forward messages. If one such higher level process crashes, all the child processes of the crashed process will not receive events until the system reconfigures to construct the dissemination tree. The independent trees (i.e., non-crossing paths between the source and the receiving processes) are used to minimize the effect of such crashes and improve the efficiency as well as the reliability of the dissemination scheme. These multiple paths can be used in different ways depending on the nature of the application and the messages being broadcast. They are further discussed later in the paper.

Example. Before continuing the elaboration of our scheme, let us consider a simple example. As shown in Figure 1, the source sends events along two separate paths (more than two paths are of course possible) for a given cluster, and hence any given process receives events along two different paths. For instance, process B in cluster S1 receives events via process A as well as C. In the case of failure (either A or C), B still receives a part or all events from one path (i.e., according to the configuration as discussed in Section 7).

When a new process joins, it must join multiple dissemination trees. For example, when G joins in Figure 1, it can select A as parent to receive events along one path and B as parent to receive events along the other path, provided that there are free out going links; if there are no outgoing links, the tree is re-arranged and the new process becomes an internal node (as described in the example of the previous section). Note that the new process G will receive messages from its own sub-cluster (from A and B) to minimize the transient traffic between sub-clusters. This reduces the message delay and congestion in links between such sub-clusters. Also, to *re-direct* G to B by process C, process C should keep some information about B and sub-cluster ID of B. These issues are described in Section 5. It is sub-optimal to have communication links between sub-clusters (like between SS1 and SS2) in these trees: but for a small number of independent trees and sufficiently large number of processes having non-zero out going links, such communication links between sub-clusters are limited.

When there is just a single process in a cluster as in S2, that process receives events on both paths from the source itself. As more processes join, the system reconfigures itself: for example, as in cluster S3. That is, in S3 process D and E exchange events that they do not receive directly from the source.

To guarantee the reliability of our scheme in the case of failures, two complementing paths should obviously not have common processes. The algorithm arranges processes according to the locality and rearranges them to make the scheme efficient as new processes join.

4.1 Reliability: An Analysis

The processes which take part in the message dissemination can be scattered all over the Internet and their behavior (in terms of join, leave and crash) is quite unpredictable. Either because the user terminates a process or due to failures, a process can be disconnected from the graph. Since many such processes act as parents, this might lead to form a disconnected tree causing inability to deliver messages to lower level processes in the tree. Of course, other trees in the graph could deliver messages and their operation is vital in such a scenario. We analyse the impact of such disconnections of trees on the reliability of the broadcast. For this, we use the following notions: 1) Mean Time Between Failures (MTBF) is the mean period of time a user may expect a given system to operate before a failure; 2) Mean Time To Recover (MTTR) is the mean period of time to recover the failed system (e.g., reconstruct a tree after disconnection). The availability of a system is defined as follows:

$$Availability\ (a) = \frac{MTBF}{MTBF + MTTR} \tag{1}$$

If the time is measured in minutes, the *down-time* per day, that is the mean time a given system is not available is :

$$Downtime = (1 -\ a) \times 24 \times 60 \tag{2}$$

Assuming the availability of a single tree is a, availability of k such trees, out of m trees in the graph, is A_k, where:

$$A_k = \binom{m}{k} a^k (1-a)^{m-k} \tag{3}$$

Hence the availability of at least k trees is α_k, where:

$$\alpha_k = \sum_{i=k}^{m} A_i = \sum_{i=k}^{m} \binom{m}{i} a^i (1-a)^{m-i} \tag{4}$$

For the proper functioning of the broadcast scheme, there should be at least one failure free tree at any given time. This tree can be used either to receive messages directly or to recover the messages (by retransmission from parent) once the message digests are received by a process (see Section 7 for more details). Availability of at least a single tree out of m trees is α_1, where:

$$\alpha_1 = \sum_{i=1}^{m} A_i = \sum_{i=1}^{m} \binom{m}{i} a^i (1-a)^{m-i} \tag{5}$$

Assume an extreme case where, for a given dissemination tree, MTBF is 15 minutes and MTTR is 4 minutes; that is a dissemination tree gets disconnected each 15 minutes on average due to a process leaving the graph and it takes 4 minutes to reconstruct the tree again (more on this is at the end of this section). Then using Equation 1 and 2 it can be seen that availability of a single tree (or any other scheme based just on a single path) is 0.7895 and the down-time is 303.15 minutes (around 5 hours) per day. Using Equation 5 and 2 it is possible to calculate the down-time of a system with m independent trees. For various values of m, the down-time per day is shown in Figure 2. For a dissemination scheme with 4 independent trees, the down-time is 2.82 minutes per day while with 6 trees it is 0.12 minutes per day.

This shows that, even in a very dynamic and unpredictable environment, where a trees fails at every 15 minutes (on average) due to processes leaving the graph, the reliability of the scheme can be improved considerably by having a few independent trees.

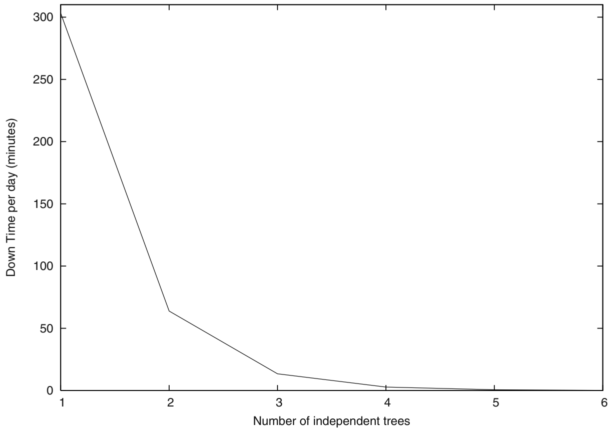


Fig. 2. Down-time of the broadcast scheme (with MTBF = 15min and MTTR = 4min)

It should be noted that the reliability can also be improved by reducing MTTR. This can be done by localizing the reconstruction: that is, whenever a process crashes or leaves the tree, the neighbors of that process reconstruct the tree without involving all the processes in the tree. The construction of the entire tree takes more time than the localized reconstruction.

5 Algorithm

In Figure 3 and 4, we describe the basic algorithm which constructs the dissemination graph. For presentation simplicity, we only show the major parts of the algorithm. When a process (potential receiver) joins a graph, the process sends a number of requests to the source (broadcaster) and then possibly to a set of other processes. It is important to note that the messages we refer to in this section are only *protocol* messages (e.g., *Join*, *SourceAccept*, *Accept*, etc.) that are used to construct the graph. They are not the actual application *broadcast* messages that is broadcast by the source. For all these *protocol* messages, the sender of a given message is denoted as P_s and the receiver is denoted as P_r.

The graph consists of m independent trees: each tree has a color to identify itself. A process receives messages from all the trees but only forwards messages from one tree, *T*. The color of a process is the color of that tree *T*. A child process that has the same color as parent p, is called a *direct* child of p. A parent has a list of all children denoted as *children_list* and list of all direct children denoted as *direct_c*. It is obvious that *direct_c* $\subset$ *children_list* for its own sub-cluster.

A process is considered as *full* if it can not have -more- outgoing links. A process which has a parent from another sub-cluster than its own, is called a *sub-cluster head*.

5.1 Basic Sequence of Messages

We describe here, the basic sequence of messages shown in the algorithm (Figure 3 and 4). When first joining, the new process, P_n, finds its country and AS (autonomous system). Then P_n sends a *Join* message to the source: the source replies either with a *SourceAccept* or *Redirect* message. If an existing process P_e (other than the source) can accept P_n as a child, then P_e sends an *Accept* message to P_n. If a process detects itself as a sub-cluster head, then it sends a *Route* message to its parent: all the processes forward this message to their parents. While sub-cluster heads alter this message before forwarding, others forward the message as it is. Once it is received by the source, the message is discarded. After receiving all m *Accept* messages, the new process P_n decides on its color: then P_n sends a *SetColor* message to its parents. The parents update their *children_list* and *direct_c* lists accordingly.

These messages (both requests and responses) and the tasks associated with them are shown in Figure 3 and 4. These request and response messages (simply stated as "messages" in this section) and associated tasks are described briefly next.

```
 1: For connecting to the broadcast group:
 2: get country c and AS as
 3: send join(c, as) message to $S$

 4: upon receiving a Join(c,as) from $P_s$ by $P_r$
 5: if $P_s$ and $P_r$ are in same AS then
 6:    if $P_r \neg full$ then
 7:       $P_r$ sends accept($|children\_list|,|direct\_c|$) to $P_s$
 8:    else
 9:       $P_r$ sends redirect(direct children of $P_r$ in same AS)
10: else
11:    if $P_r$ is a sub-cluster-head then
12:       if $\exists P_e \in$ routing_table such that $P_e$.AS=as then
13:          $P_r$ sends redirect($P_e$) message to $P_s$
14:          else Do as 15 to 18
15:    else if $P_r \neg$ full then
16:       $P_r$ sends accept($|children\_list|,|direct\_c|$) to $P_s$
17:    else
18:       $P_r$ sends redirects(direct children of $P_r$) to $P_s$

19: upon receiving a accept($|$children_list$|$,$|$direct_c$|$)
20: set $P_r$.parent =$P_s$
21: if all $m$ accept messages are received then
22:    if parent $P_i$ of $P_r$ is from another sub-cluster then
23:       select color of $P_i$
24:       $P_r$ sends Route($P_r$_id,AS of $P_r$)
25:    else
26:       select a parent who needs more direct children {using
          parameter of the message}
27:    $P_r$ sends SetColor($P_r$_id,color of $P_r$) to $P_s$

28: upon receiving a Redirect(list)
29: let $P$ = set of potential parents
30: if $P_s$ is the source then
31:    for all $p \in$ list do
32:       send join(c,as) messages
33: else
34:    $P \leftarrow P \cup$ list
35:    $t \leftarrow$ select one from $P$
36:    $P \leftarrow P \setminus t$
37:    send join(c,as) message to $t$

38: upon receiving a SourceAccept(list, clr)
39: set color to clr
40: for all $p \in$ list do
41:    send join messages
```

Fig. 3. Graph Building Algorithm: At every receiver process

```
upon receiving a Route(process_id, p_AS)
if $P_r$ is a sub-cluster-head
    routing_table← (process_id, p_AS)
    forward Route($P_r$_id, p_AS) to parent of $P_r$
else
    forward Route(process_id, p_AS) to parent of $P_r$

upon receiving a SetColor(process_id, clr)
$children\_list$← $children\_list$ ∪ (process_id, clr)
```

Fig. 3. (*Continued*)

5.2 Messages

Join This message is sent by P_s to P_r when a process P_s needs to join a group. First the message is sent to the source. Then, depending on the response this message will be sent to other processes. The source might respond with a *SourceAccept* or *Redirect* message. Other processes respond with an *Accept* message or *Redirect* message. If the processes (not source) P_s and P_r are not in the same sub-cluster and P_r is a sub-cluster head, then P_r performs a routing table lookup in *routing_table* to find another process that is in the same sub-cluster as of P_s.

SourceAccept. This message is sent by the source (as a response to a *join* message) when the source decides to accept a receiver process as its own child. That is, when there are less than m processes in a cluster C (country) at the bootstrapping (initial) phase, these messages are sent to construct the initial set of children and the graph. The information about the process ids (of source's children in the country C) and their corresponding color is also sent along with this message. Then each process sets its color according to the request of the source and sends a *join* message to all other processes as indicated by *process_ids*.

```
upon receiving a Join(c,as) from $P_s$
let $C$=immediate children of $S$ in country c
if $|C| < m$ then
    $C$← $C \cup P_s$
    for all $p \in C$ do
        clr← select color for $p$
        send SourceAccept($C$,clr) message
else
    send redirect($C$) to $P_s$
```

Fig. 4. Graph Building Algorithm: At the source S

Accept. This message is sent by a process P_s to another process P_r as a response to a *join* message when P_s decides to accept P_r as a child. If one parent is from another sub-cluster, P_r selects the color of this parent. As a result, subsequent processes from the same cluster (as of P_r) can find a parent from within their own cluster. This reduces the number of links between the sub-clusters. Since such links are associated with greater delays (than links in the same cluster), by reducing such links, the efficiency is increased. The two parameters (number of elements in *children_list* and in *direct_c*) allow a child to decide its color in an efficient manner. Using this parameter it is possible to estimate whether one parent is not having adequate direct children (i.e., enough out-going links) of its own color. Then P_r can select the color of the process which does not have adequate direct children. In other terms, this selection criterion helps to have enough out-going links of each color.

Redirect. This message is sent by P_s to P_r in response to a *join* message sent by P_r (who is looking for a parent). The parameter, *list*, is a set of possible parents to P_r. If P_s is the source, then ($|\text{list}| = m$) P_r will send *m* number of *join* messages to construct *m* trees. If P_s is not the source, then P_r explores each process in “list” to find a suitable parent by sending *join* messages to them. This join-redirect set of messages resembles a search in a tree to find a possible parent.

Route. This message is used to update the routing information in sub-cluster heads, (a sub-cluster head is a process whose parent comes from a different sub-cluster than his own). These processes keep a small amount of routing information. As a result, given a process from a particular sub-cluster AS_i, the sub-cluster head knows whether there are any other processes in this AS_i, in the sub-tree below the sub-cluster head. If such a process exists, a new process which tries to join can be redirected appropriately. In short, this message helps to group processes of the same sub-cluster together. This message has a parameter $<process\_id, p\_AS>$, where p_AS is the sub-cluster which could be reached via $process\_id$. Sub-cluster heads alter the $process\_id$ and forwards the new *route* message to their parents.

SetColor. This message is sent to P_r by P_s in response to an *accept* message indicating that P_r is selected as the parent. The parameter “color” indicates the color of P_s. P_r keeps information about ID of P_s and color of P_s.

For the presentation simplicity, the Figure 3 and 4 show only the major parts of the algorithm. The procedure of broadcasting (application message forwarding) is not shown in the algorithm; but this is simply done by forwarding messages along the edges of the trees by each process starting from source. The information a process stores (e.g., *children_list*) has the nature “soft state”; that is, these informations need to be refreshed periodically (e.g., in this case by children). In other terms, in order to be in the *children_list*, children need to inform parents about their presence periodically. This nature of “soft state” information enables to handle crashes and leaving of processes without any notification.

When a process (in particular an intermediate node in a tree) crashes (or leaves without any notification), one of the path (of a given color) in the dissemination graph is broken. Under such circumstances (once the children observe the crash of parent), after a timeout, the children of crashed process should initiate a procedure to reconstruct the broken path. The naive approach would be that these children contact the source again to construct a path with the color of the broken one (other paths are unaffected). This naive solution is not optimal in terms of communication steps, but consumes less memory as it does not require additional information for the reconstruction procedure.

6 Performance

In this section we present results obtained from 1) a prototype implementation as well as from 2) simulations. We use the prototype to test the feasibility of BGP-based clustering in a real world scenario as well as to evaluate the performance of the dissemination scheme. The feasibility of clustering method in a larger scale is also tested using a simulation. We also use simulation to check the performance of the system with different values of fanout.

6.1 Prototype Implementation

In this real experiment, we used a source which broadcasts messages with the size of 1kb each to a total of 210 processes.

Setting. A source publishes over a modem connection (56kbps) from country A and in autonomous system X. We used 150 processes in one country (A), and in autonomous system Y, while 60 other processes were in another country (B), and in the same autonomous system. A random value between 1 to 4 was chosen as the "fanout" (i.e., max. number of outgoing links) for each process. This simulates a real-life setting where the fanout of an individual process depends on its network bandwidth and user preferences. As described in Section 3, the country is specified for each process and the autonomous system is obtained using the WHOIS [21] service.

Results. Table 1 summarizes the results obtained in this experiment. The details of these measurements are discussed next.

Table 1. Communication between two Clusters

Country	A	B
Number of processes	150	60
Maximum depth	8	8
Average depth	5.3	5.7
Maximum join latency (ms)	1758	1302
Average join latency (ms)	952	1150
Minimum join latency (ms)	686	757
Maximum delay (ms)	213	267
Average delay (ms)	195	258
Minimum delay (ms)	177	182

Maximum and Average Depth: The depth of a process reflects the number of hops taken by a message before being delivered to that process. The maximum time taken to disseminate a message depends on the maximum depth of the dissemination tree used for its dissemination. The average depth in contrast is the average number of hops taken by messages before being received by processes.

Maximum and Minimum Join Latency: In the dissemination scheme, a process joins a suitable parent in the join phase. As processes are redirected progressively starting from the source (see Section 3.2) to other processes, there is a certain latency when joining the dissemination tree. The maximum and minimum latencies are depicted by these two values.

Maximum, Average, and Minimum Propagation Delay: As each message is forwarded a given number of times by the processes, there is a delay before a message is received by each process. The maximum propagation delay is the largest delay incurred when receiving a message in the system. This delay occurs for the bottom most process in the dissemination tree. Similarly, minimum and average propagation delay represents the minimum and average delay incurred in the dissemination process.

6.2 Simulations

We performed a set of simulations to analyze the performance of our clustering scheme beyond the above (admittedly limited) setting involving only 2 countries and 3 autonomous systems. We were interested in finding 1) the maximum delay (in terms of hops) incurred due to successive forwarding of messages between processes, and 2) the impact of the fanout on the maximum delay.

Setting. We simulated a set of IP clients which are globally distributed. To achieve this, we used a set of IP addresses of hosts which had recently accessed our laboratory web site. To group IP addresses into clusters and sub-clusters within each country, we used the WHOIS [21] service from [22].

We associated, -with each IP address-, a random integer f such that f is bounded by $1 \leqslant f \leqslant k$. The parameter f depicts the fanout or the number of out-going links from a process to other processes. We varied k such that k= 2,3,4,5 and did the simulations for each k. As a result, we represent processes which are capable of forwarding messages to up to k other processes as well as ones that can forward messages to only 1 other process.

Results. For each value of k we constructed the dissemination tree as shown in Figure 1 (Y-axis = depth, X-axis = fanout) and found the maximum and average depth of the tree to estimate the maximum delay incurred due to hops when disseminating the messages. Since the delay incurred for each message depends on the number of hops the message has, it is critical to limit the number of hops. The maximum number of hops a message will have is equal to the maximum depth of the tree in our dissemination scheme. Since it is more general and appropriate to express the delay in terms of hops in end processes based (peer-based) systems, we used hops as the measure of the delay in our results.

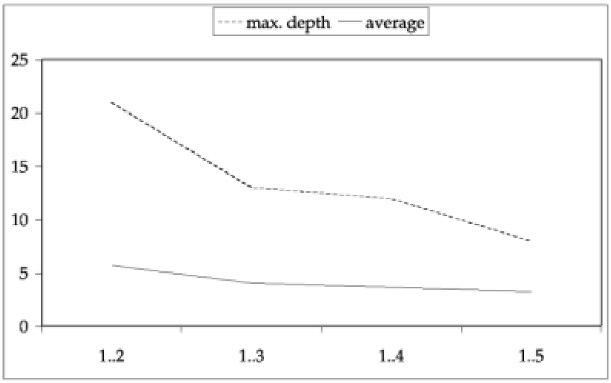


Fig. 5. Maximum Delay for Different Values of k

Figure 5 shows the delay for each value of k. Here, -for example- when k=2, the fanout of processes can have a value of 1 or 2.

7 Applications

In this section we describe two specific applications that can be built on top of our broadcast. First, we present a scheme suitable for streaming media. Second, we describe a general gossip-based broadcast that will reduce the network usage (by using message digests) while preserving the necessary redundancy to handle message losses and failures.

7.1 Broadcasting Stream Data

Stream data such as audio and video consists of samples which are broadcast in a fragmented fashion. These fragments, which form a "signal" (e.g., a video frame, an audio clip), are re-assembled at the receiver. Fragmentation can be done such that even if a set of fragments are lost, this does not necessarily invalidate an entire message (e.g., a video frame, an audio clip). There are number of coding schemes (e.g., [24, 25]) which can deliver adequate quality stream data in spite of high level of message loss. Our independent dissemination trees can use such coding schemes very efficiently to deliver stream data. These successive (or interleaving) fragments (packets) can be routed in a round-robin style over multiple trees.

In such a scenario, the crash of a path does not cause a complete loss of the broadcast. The crash of a path only degrades the quality of the signal until that path is reconstructed. This technique is hence applicable whenever the composition of single events/messages generates higher quality aggregated data.

In the context of streams, our scheme applies particularly well since the redundant trees are efficiently used: that is, the redundant trees do not disseminate duplicate messages but messages that can augment other messages.

7.2 Deterministic Gossip

The independent trees of the dissemination graph can be used to implement gossip-based dissemination scheme in a deterministic fashion. In other terms, as the simplest case, processes receive the same set of messages via independent and redundant trees (e.g., m different trees). The source can set the parameter m according to the level of redundancy required by the application to circumvent message losses and process failures.

Another approach is to send messages in k (where $k < m$; e.g., $k = 1$) trees and use other $m-k$ trees to send digests (i.e., message IDs) of those messages. As a result, when the system operates without process failures and message losses, a process receives actual messages k times and message digests from $m-k$ trees. In the case of k trees that send actual messages fail, the process still receives message digests from other independent trees. Under such circumstances, a process is aware that it is not receiving all messages that are being broadcast. Then (after a time-out) the process can ask for messages from its parent in the correctly functioning trees from which it receives the digest of the actual message. This recovery phase of messages is efficient since it is done using neighbors of a given process (localized recovery) instead of using the source itself. This method of deterministic gossip could help to reduce the amount of network usage while still maintaining good reliability properties by having redundancy.

8 Conclusion

This paper presents a scalable gossip broadcast algorithm with good reliability properties. Broadcast is achieved using a graph, consisting of processes grouped according to their locality. Processes (including the broadcaster) forward messages to a limited number of other neighbors. This number is defined according to their capabilities in terms of resources. The processes only know about limited number of other processes.

To group processes according to their locality, a clustering scheme based on BGP information is used. This scheme arranges processes in the Internet according to their “locality”. Consequently, message delays between processes and transient broadcast traffic between large networks (autonomous systems) are reduced by localizing the majority of broadcast traffic within clusters and sub-clusters. The clustering scheme, together with the local knowledge of processes, help new processes find a suitable “place” within the graph by using few communication steps.

The clustering approach we use to arrange processes can be applied to various applications (e.g., peer-to-peer applications) and other broadcast algorithms such as [26, 27, 18]. Our dissemination graph with multiple independent paths is particularly suitable for broadcasting streaming data such as audio and video media.

The processes self-organize to construct the graph which consists of “non-crossing” (independent) trees. These trees, which evolve in a dynamic environment, are used to forward messages. As shown in the paper, even in extreme cases where processes leave the dissemination graph often, it is possible to have good reliability properties by limiting the down-time to a required level. We also convey the feasibility of our approach both using simulations and experimental results.

References

1. Hadzilacos, V., Toueg, S.: 5: Fault-Tolerant Broadcasts and Related Problems. In: Distributed Systems. 2nd edn. Addison-Wesley (1993) 97–145
2. Deering, S.: Host extensions for IP multicasting; RFC 1112. Internet Requests for Comments (1989)
3. Eriksson, H.: Mbone: The multicast backbone. Communications of the ACM **37** (1994)
4. hua Chu, Y., Rao, S.G., Seshan, S., Zhang, H.: A case for end system multicast. In: IEEE Journal on Selected Areas in Communication (JSAC), Special Issue on Networking Support for Multicast. (2002)
5. Francis, P.: Yoid: Extending the internet multicast architecture. http://www.isi.edu/div7/yoid/docs/index.html (2000)
6. Chawathe, Y.: Scattercast: An adaptable broadcast distribution framework. In: Special issue of the ACM Multimedia Systems Journal on Multimedia Distribution. (2002)
7. Deshpande, H., Bawa, M., Garcia-Molina, H.: Streaming live media over a peer-to-peer network. http://dbpubs.stanford.edu/pub/2002-21 (2002)
8. Birman, K., Hayden, M., O.Ozkasap, Xiao, Z., Budiu, M., Minsky, Y.: Bimodal multicast. ACM Transactions on Computer Systems **17** (1999) 41–88
9. Li, Z., Mohapatra, P.: Hostcast: A new overlay multicasting protocol. In: Proceedings of the IEEE International Communications Conference (ICC). (2003)
10. Castro, M., Jones, M., Kermarrec, A.M., Rowstron, A., Theimer, M., Wang, H., Wolman, A.: An evaluation of scalable application-level multicast built using peer-to-peer overlay networks. In: Proceedings of the IEEE Conference on Computer Communications (INFOCOM). (2003)
11. Francis, P., Jamin, S., Jin, C., Jin, Y., Raz, D., Shavitt, Y., Zhang, L.: Idmaps: A global internet host distance estimation service. In: IEEE/ACM Trans. on Networking, Oct. 2001. (2001)
12. Theilmann, W., Rothermel, K.: Dynamic distance maps of the internet. In: Proceedings of the IEEE Conference on Computer Communications (INFOCOM). (2000)
13. Lin, M.J., Marzullo, K.: Directional gossip: Gossip in a wide area network. In: Proceedings of European Dependable Computing Conference (EDCC). (1999) 364–379
14. Rekhter, Y., Li, T.: A border gateway protocol 4 (bgp-4). RFC-1771, http://www.ietf.org/rfc/rfc1771.txt (1995)
15. L.Garces-Erice, Ross, K.W., Biersack, E.W., Felber, P.A., Urvoy-Keller, G.: Topology-centric look-up service. In: Proceedings of COST264/ACM Fifth International Workshop on Networked Group Communications (NGC). (2003)
16. Deering, S.: Multicast routing in internetworks and extended lans. In: Proceedings of ACM SIGCOMM. (1988)
17. Eugster, P.T., Guerraoui, R., Handurukande, S.B., Kermarrec, A.M., Kouznetsov, P.: Lightweight probabilistic broadcast. In: Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN 2001). (2001)
18. Gupta, I., Kermarrec, A.M., Ganesh, A.: Adaptive and efficient epidemic-style protocols for reliable and scalable multicast. In: Proceedings of 20th Symposium on Reliable and Distributed Systems (SRDS 2002). (2002)
19. Xiao, Z., Birman, K.: Randomized error recovery algorithm for reliable multicast. In: Proceedings of the IEEE Conference on Computer Communications (INFOCOM). (2001)
20. Castro, M., Druschel, P., Kermarrec, A.M., Nandi, A., Rowstron, A., Singh, A.: Splitstream: High-bandwidth multicast in a cooperative environment. In: Proceedings of The ACM Symposium on Operating Systems Principles (SOSP). (2003)
21. Harrenstien, K., Stahl, M., Feinler, E.: Rfc 954: Nicname/whois. http://www.rfc-editor.org/rfc/rfc954.txt (1985)

22. Web, M.: The routing arbiter project. http://www.ra.net/ (2002)
23. Bourcier, P.: Cyberabuse. whois.cyberabuse.org (2001)
24. Cai, J., Chen, C.W.: Fec-based video streaming over packet loss networks with pre-interleaving. In: Proceedings of IEEE International Conference on Information Technology: Coding and Computing (ITCC '01). (2001)
25. Leslie, B., Sandler, M.: Packet Loss Resilient, Scalable Audio Compression and Streaming for Wired and Wireless IP Networks (White Paper). (2002)
26. van Renesse, R., Birman, K., Vogels, W.: Astrolabe: A robust and scalable technology for distributed systems monitoring, management, and data mining. ACM Transactions on Computer Systems **21** (2003)
27. Eugster, P.T., Guerraoui, R.: Probabilistic multicast. In: IEEE International Conference on Dependable Systems and Networks (DSN 2002). (2002)

Trust Lifecycle Management in a Global Computing Environment

S. Terzis, W. Wagealla, C. English, and P. Nixon

The Global and Pervasive Computing Group,
Dept. of Computer and Information Sciences,
University of Strathclyde

Abstract. In a global computing environment in order for entities to collaborate, they should be able to make autonomous access control decisions with partial information about their potential collaborators. The SECURE project addresses this requirement by using trust as the mechanism for managing risks and uncertainty. This paper describes how trust lifecycle management, a procedure of collecting and processing evidence, is used by the SECURE collaboration model. Particular emphasis is placed on the processing of the evidence and the notion of *attraction*. *Attraction* considers the effects of evidence about the behaviour of a particular principal on its current trust value both in terms of trustworthiness and certainty and is one of the distinctive characteristics of the SECURE collaboration making it more appropriate for a global computing setting.

1 Motivation

Global computing is characterised by large numbers of roaming entities and the absence of a globally available fixed infrastructure [33]. In such an environment entities meet and need to collaborate with little known or even unknown entities. Entering any kind of collaboration requires entities to make security decisions about the type and level of access to their resources they will provide to their collaborators. In traditional environments with clearly defined administrative boundaries and limited entity movement security decisions are usually delegated to a centralised administrative authority [34, 25, 26]. In the global computing environment no single entity can play this role and as a result traditional techniques that statically determine the access rights of the entities are not an option. Entities are required to make their own security decisions. Moreover, the absence of a globally available security infrastructure means that these decisions need to be made autonomously. At the same time the sheer number of the roaming entities means that it is not feasible to gather and maintain information about all of them. Consequently, in the global computing environment decisions have to be made in the absence of complete knowledge of the operating environment.

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 291–313, 2004.

The SECURE[1] project [35] recognises that the characteristics of the global computing environment mean that any effort to provide absolute protection of roaming entities against potential dangers, as envisioned by traditional security approaches, is not feasible. It also observes that autonomous decision making with partial information is something that humans have to deal with on a day-to-day basis. To help them with the complexity of such a task humans have developed the notion of trust [16]. Although trust is an elusive concept and a number of definitions have been proposed for it, the SECURE project believes that it can be modelled in adequate detail to facilitate security decision making in global computing. As a result, it takes an approach to security that accepts that dangers are an intrinsic part of the global computing environment and uses trust as a mechanism for managing these dangers/risk. Central to such an approach is the support for explicit reasoning about risk and uncertainty.

The potential advantages of the notion of trust in dealing with security decisions have been recognised by a number of researchers as is demonstrated by research in Trust Management systems [1, 4, 5, 11, 20, 22, 28, 37, 38]. Although this work is a move forward in security practice, most of it is based on the exchange of certificates between entities [4, 5, 20, 22, 28], and does not address the fundamental issue of what trust is made of and the related issue of how trust can be formed. At the same time, it provides very limited support for the evolution of trust between entities in the form of certificate revocation. Furthermore, this approach does not really consider risk and uncertainty. As a result, this work lacks the support for autonomous decision-making and for dynamism in trust evolution necessary for global computing. The SECURE project addresses these limitations by providing the following:

- A trust model with an explicit notion of uncertainty [7, 9, 10].
- A risk model [3, 13, 30].
- A collaboration model [15, 32, 14], which combines the trust and risk and addresses trust formation and evolution as learning from past interactions processes aiming to improve entity protection.

In this paper, we focus on how the trust lifecycle is managed within the SECURE project and in particular how evidence about the past behaviour of entities is processed. To further demonstrate our approach we also describe as an example how trust lifecycle management could be carried out in an e-purse application scenario. We start our presentation with a brief outline of the SECURE collaboration model, starting from its foundations, the trust and risk models, and its basic processes, decision-making, trust evaluation and risk evaluation. We then focus on the types of evidence considered and how they are processed, followed by a discussion on the formation of trust and the presentation of the e-purse application scenario. We finally compare our approach to the state of art and provide our conclusions and future work.

[1] SECURE (Secure Collaboration among Ubiquitous Roaming Entities, IST-2001-32486) is an EU FET Research Project funded under the Global Computing Initiative.

2 The SECURE Collaboration Model

The aim of the collaboration model is to capture the dynamic aspects of the trust model. These aspects address issues like how trust is formed, how it evolves over time and how it is exploited in the access control decision making process. The model is founded upon the trust and the risk model. In particular, it exploits the relationships between trust and risk to both facilitate and evaluate the access control decision making process.

We define *collaboration* as a joint interaction between a set of two or more principals $\mathcal{P}$ involving a set of one or more trust mediated actions $\mathcal{A}$. Before entering a collaboration each principal must make an access control decision regarding the level of access to its resources it will permit to other principals.

In order for the terminology of the above definition to become clear, we start this section with a brief presentation of the underlying trust and risk models that leads on to the description of the three processes supported by the model, namely access control decision making, trust evaluation and risk evaluation. All three processes rely on the processing of evidence about principals' behaviour. As a result, evidence processing is at the heart of managing the trust lifecycle as it determines both how trust is formed and evolves over time. A detailed discussion of the other aspects of the SECURE collaboration model can be found in [32].

2.1 Trust and Risk

The trust model [7, 9, 10] considers principals to be entities that either have to make trusting decisions or are the subjects of these decisions. $\mathcal{P}$ is defined as the set of all principals. Trust reasoning for principals has two aspects. On one hand decision making principals should be able to associate trust values to other principals. $\mathcal{T}$ is defined as the set of all these trust values. On the other hand, principals should also be able to update their trust values in the light of evidence. Consequently, given the set of principals $\mathcal{P}$ and the set of trust values $\mathcal{T}$ the global trust is defined as a function $m : \mathcal{P} \rightarrow \mathcal{P} \rightarrow \mathcal{T}$, where $m(a)(b) \in \mathcal{T}$ expresses a's trust in b.

Following an object-based model, the ability of each principal to reason about trust is modelled as a trust box, which has some internal trust state S and supports two operations:

- `update`: $\mathcal{S} \times \mathcal{E} \longrightarrow \mathcal{S}$, given a particular trust state $\mathcal{S}$ and some evidence $\mathcal{E}$, an updated trust state is produced.
- `trust`: $\mathcal{S} \times \mathcal{P} \longrightarrow \mathcal{T}$, given a particular state $\mathcal{S}$ and a principal $\mathcal{P}$, the trust value for the principal is returned.

The operation of each principal's trust box is described by a local policy function π, which relates principals to trust values and supports references. References provide the ability for a principal to specify trust values as relations over the trust values of other principals. This local policy π is defined as:

$$\pi : (\mathcal{P} \rightarrow \mathcal{P} \rightarrow \mathcal{T}) \rightarrow \mathcal{P} \rightarrow \mathcal{T}. \tag{1}$$

The collection of all the local policies defines a global trust policy:

$$\Pi : (\mathcal{P} \to \mathcal{P} \to \mathcal{T}) \to (\mathcal{P} \to \mathcal{P} \to \mathcal{T}). \tag{2}$$

This global trust policy is interpreted in terms of complete partial orders. If the set of trust values $\mathcal{T}$ given an ordering relation $\sqsubseteq$ is a complete partial order (c.p.o.) with a least element $\perp$ (unknown), then the global trust m can be calculated as the least-fixed point of the global trust function Π. The ordering relation $\sqsubseteq$ represents the level of certainty in the trust values and is therefore referred to as *certainty ordering.*

In addition to the $\sqsubseteq$, "more certainty" ordering relation on $\mathcal{T}$, the model also defines $\preceq$, "more trust", which is equally essential. The $\preceq$ relation specifies for two trust values t_1 and t_2, which one expresses more trust. According to the theoretical trust model the set of trust values $\mathcal{T}$ given the ordering $\preceq$ is a complete lattice.

The risk model [3, 13, 30] considers each collaboration between principals as consisting of a number of trust mediated actions. Each such action is an interaction between two principals P_r and P_d, the *requester* and the *decision-maker* respectively, and has a set of possible results or outcomes. Each outcome has an associated risk. Risk is defined as the likelihood of an outcome occurring and the cost or benefit this outcome incurs if it occurs. The risk of an outcome depends on the trustworthiness of the requester P_r and certain parameters of the action in question. Before each trust mediated action the decision-maker P_d must make the decision of whether to trust the requester to carry out the requested action. This decision is based on the overall risk of the action in question, which is a combination of the risks of all its outcomes. We refer to the overall risk that a particular action with a particular principal entails as the *risk profile* of the principal.

Although others have also recognised the importance of explicit modelling of both trust and risk (e.g. [12, 17]), the SECURE project is unique in defining a clear relationship between trust and risk. Within the SECURE project we take the view that the trustworthiness of the requester P_r determines the likelihood of the various outcomes, while their associated costs or benefits are determined by the parameters of the action. For example, in the case of financial transactions the trustworthiness of the principals determines the chances of them paying their debts, while the debt amount determines the specific costs or benefits. Moreover, we take the view that the relationship between trust and risk dictates that more trustworthy principals make beneficial outcomes more likely and/or costly outcomes less likely. While, as the certainty in the principal's trustworthiness increases the probability mass is more concentrated towards certain outcomes. As a result, the trust value for principal, taking into account both its trustworthiness and certainty dimension, determines the risk profile for a particular action involving this principal.

2.2 Collaboration Model Processes

Having outlined the foundations of the SECURE collaboration model, we can proceed to describe the processes comprising the model. As we have already mentioned before these processes are: (a) access control decision making, (b) trust evaluation and (c) risk evaluation. The aim of the three processes is to support both decision making and evaluation of these decisions in the context of both trust and risk.

The collaboration model takes the view that trust drives the access control decision making and the decision that the decision making principal has to make can be expressed as: *for a particular situation st, or a particular action a, involving a particular principal p, how much risk are we willing to accept by allowing principal p to enter situation st or carry out action a?*

In this context, the access control decision making process includes the following steps:

1. **Collaboration Request.** This step is the triggering of the decision making process. There are two alternative ways in which the decision making process may be initiated. Either the decision maker receives a request for a particular action from a requester, or as a requester it needs to decide which principal to approach with a request for a particular action. Note that in the latter case there is not really a collaboration request as such.
2. **Principal Recognition.** In this step irrespective of how the decision making process was triggered the decision maker needs to determine, who are the principals involved. Principal recognition is a superset of authentication that does not require the pre-registration of principals [29]. As a result, it is better suited for a global computing environment allowing the collaboration with previously unknown principals.
3. **Principal Trust Assignment.** Having identified the principal in question then the decision maker can apply its local trust policy in order to determine their trust value. Note that this process in the case of unknown principal will return the $\perp$ ("unknown") trust value, the bottom element of trust certainty ordering (see section 2.1).
4. **Collaboration Risk Assessment.** The trust value for the principal in question provided by the previous step can now be used to determine the principal's risk profile for the particular action. Note that the exact nature of this step depends on the way in which the trust and risk domains for the particular application have been defined. In any case this process is where the relationship between trust and risk is exploited.
5. **Access Control Policy Application.** In the final step of the decision making process the access control policy of the decision maker is applied to reach a decision. The access control policy is in terms of risks entailed and determines which risks are (un)acceptable for the decision maker. We should point out that the decision is not necessarily binary, i.e. accept/reject.

During interaction with others principals collect information about their behaviour. As this information becomes available each principal needs to evaluate its decision making. The evaluation is twofold. On one hand, the principal needs to evaluate whether the trust value for each principal is correct. We refer to this process as *trust evaluation*. This process involves the processing of collected evidence regarding the behaviour of the same principal over a number of different instances of the same action. On the other hand, the principal also needs to evaluate whether the risk profiles used are correct. In other words, it need to evaluate whether the cost and benefits associated with each outcome of each action are correct. We refer to this process as *risk evaluation.* This process involves the processing of collected evidence regarding the behaviour of multiple principals over a number of instances of the same action.

Trust evaluation involves the following steps:

1. **Collaboration Monitoring.** In this step the behaviour of principals is monitoring during a collaboration. Evidence regarding the occurred outcome of action is produced.
2. **Evidence Processing.** In this step the evidence collected about the behaviour of principals is processed. We should point out that the collected evidence is not just from personal interactions with a principal but also from other principals' interactions with the principal in question. This step is elaborated in the following section.
3. **Update Principal's Trust Value.** In the final step the processed evidence is used to determine if it is necessary to change the current trust value for the principal and what is the most appropriate new trust value.

Risk evaluation involves the following steps:

1. **Collaboration Monitoring.** This step is very similar to the collaboration monitoring step in trust evaluation. The only difference is in the type of evidence produced. In the case of risk evaluation we are interested in evidence regarding the incurred costs or benefits of occurred outcomes and not the outcomes themselves.
2. **Evidence Processing.** This process is quite different from the processing of evidence regarding principals' behaviour.
3. **Update Outcome Cost/Benefits.** In the final step the processed evidence is used to determine if it is necessary to change the currently used outcome cost/benefits and what is the most appropriate value.

2.3 The Nature of Evidence and Its Processing

Having described the various processes of the SECURE collaboration model, in this section we turn our attention to the processing of evidence about principals' behaviour, a core part of the trust evaluation process. However, before we can examine how the processing is carried out it is imperative to examine the exact nature and characteristics of the evidence we consider.

The Nature of Evidence. In general, evidence refers to any kind of information about principals' past behaviour. We call the principal in question the *subject* of the evidence. Evidence can be characterised as either *direct* or *indirect.* The former refers to evidence about a subject's behaviour that has been directly witnessed by a principal, the *witness.* The latter refers to evidence about a subject's behaviour that has not been directly witnessed. In fact, it refers to third party information about a subject's behaviour. From these definitions it should be clear that the characterisation of evidence as direct or indirect is relative, in other words direct evidence propagated to another principal becomes indirect.

The distinction between direct and indirect evidence is quite important. The validity of direct evidence is unquestionable and should be treated as fact, while indirect evidence can be questionable and should be treated as an opinion about facts. Consequently the validity of indirect evidence depends heavily on the source of the information, the principal expressing the opinion. This difference suggests that during trust evolution indirect evidence must be treated with caution, as different principals may evaluate differently the same facts. Moreover, some principals may even distort the facts.

The trust management literature identifies three types of evidence:

1. **Observations**. They refer to direct evidence. They are personal experiences usually gathered through interaction with a principal. Within the SECURE collaboration model, an interaction between two principals is in the form of trust-mediated actions. Each of these actions has a set of possible outcomes, each with its own costs or benefits (see section 2.1). Since, the trustworthiness of the principal determines the likelihood of each outcome occurring, we take the view that an observation is in fact the outcome that occurred at the end of the interaction.
2. **Recommendations**. They refer to indirect evidence passed between a principal W, the witness, and a principal R, the receiver, describing a judgement on principal $Subj$, the subject. In the general case, this evidence can take any form, but in this document we only consider the case of a trust value expressing W's trust in $Subj$.
3. **Reputation**. It refers also to indirect evidence that takes the form of a measure of the overall trustworthiness of a subject $Subj$. This measure can be expressed as:
$$r(Subj) = \sum_{P_i \in C} m(P_i)(Subj) \tag{3}$$
Note that we only assume a community of principals C, which is a subset of the whole principal population $\mathcal{P}$. This is in line with the lack of complete information characterising global computing.

At this point we should note that both recommendations and reputation are based on the ability of principals to exchange trust values. For any such exchange to be meaningful the trust values need to share a common representation and all principals need to have a shared understanding of their meaning. Consequently, it should be clear that a shared structure for the trust value domain is the minimum requirement for the meaningful exchange of trust values between principals. Hence, in the SECURE collaboration model we take the view that all principals share the same structure of the trust value domain, which means the following:

1. There is a single format of trust values.
2. The value range of the trust values is the same for all principals.

3. The trust and certainty orderings are the same for all principals. So, if $t_1 \preceq t_2$ and $t_1 \sqsubseteq t_2$ for principal P_i then $t_1 \preceq t_2$ and $t_1 \sqsubseteq t_2$ for all other principals.

The above assumptions guarantee to a certain degree meaningful exchange of evidence in the form of trust values between principals within a particular application.

The three different types of evidence, although all valuable for trust formation and evolution, do not have the same value. In general, putting aside considerations regarding the freshness of evidence, we would expect direct evidence to be a lot more valuable than indirect evidence due to its unquestionable character. This means that we would consider observations to be the most valuable type of evidence and as a result to carry the most weight in the evolution process. In fact, we could even see situations where observations would be the only type of evidence considered. The problem with observations is that they require the participation of the witness, making them the most difficult type to collect. It usually takes a lot of time before any principal acquires adequate personal experience.

The exchange of experiences between principals can enhance their perception of the world, especially in cases where personal experience is limited. In these cases recommendations and reputation can be particularly valuable. However, their value is predicated on the assumption that the subject is likely to behave similarly towards both the witness and the receiver. If the assumption does not hold, then the exchanged experiences are worthless. In the general case, there are no guarantees that this assumption holds. Identifying which principals witness similar behaviour from certain subjects is in most cases very difficult. In any case, the fact that exchanged experiences are indirect types of evidence means that the trustworthiness of their source affects their validity and as a result their value. This is acknowledged in the literature and has led to the introduction of the concept of discounting for recommendations [19, 21, 36]. Discounting usually takes the form of an operator which considers the receiver's perception of the integrity or trustworthiness of the witness, namely the *trust in the recommender.* We should note here that this type of trust is usually considered as separate from the trust that is normally associated to principals. It reflects how good the principal is as a source of recommendations, rather than how likely it is to behave well.

Taking the above into account the SECURE collaboration model incorporates a notion of *trust in the recommender* and introduces a *recommendation adjustment* stage in the processing of indirect evidence. This process takes place before the evaluation of the recommendations (see next section) and is along similar lines to the notion of semantic distance between recommendation and experience as defined in [1]. However, we should note that although the use of an adjusting operator that takes into consideration the trust in the recommender may be used to increase the value of recommendations, in the case of reputation things are even more complicated. Reputation aggregates a number of recommendations in order to follow the same approach we need to adjust each constituent recommendation separately. This requires that we know exactly which opinions were combined to produce the measure of overall trustworthiness, who were the sources of these opinions and how much each of them contributed to the overall measure. If this is the case, then reputation can be treated in a similar manner to individual recommendations. However, in most cases, this level of detailed information is not available, and as such reputation is not very valuable for evolution. This is the main reason why reputation is not considered in the SECURE project.

Evidence Processing. Evidence processing is carried out in two stages. First, the evidence is evaluated with respect to the current trust values for the principal in question. The aim of the evaluation is to determine whether the associated risk profile for current trust value of the principal is in accordance to the observed behaviour of the principal. For this purpose we introduce the notion of *attraction*. As a result the evidence evaluation stage is in fact where the calculation of the attraction of the evidence takes place. Second, the evaluation of the evidence, i.e. its attraction, is used to evolve the current trust value. The evolution is towards a new trust value whose associated risk profile better matches the principal's observed behaviour.

More specifically, we can view the evidence evaluation process as a function, which assuming that *Evd* is a set of pieces of evidence, and *Attr* is the set of attractions, is defined as follows:

$$evaluate : Evd \times \mathcal{T} \rightarrow Attr \tag{4}$$

Since, according to the theoretical trust model our trust values reflect both trust and certainty, we can express the impact of attraction in both trust and certainty terms. Therefore, we can view attraction as a two-dimensional measure consisting of a trust dimension (τ) and an certainty dimension (σ). On each dimension attraction can be characterised:

- In the certainty dimension as either *reinforcing* or *contradicting*. In the former case, the new evidence cannot increase the certainty of the current trust value, i.e. $T_{curr} \sqsubseteq T_{new}$. In the latter case, the new evidence cannot reduce the certainty of the current trust value, $T_{new} \sqsubseteq T_{curr}$.
- In the trust dimension as either *positive* or *negative*. In the former case, the new evidence cannot reduce the trustworthiness of the current trust value, $T_{curr} \preceq T_{new}$. In the latter case, the new evidence cannot increase the trustworthiness of the current trust value, $T_{new} \preceq T_{curr}$.

We refer to the above characterisation of attraction as the *direction of the attraction*. The reason for this is that if we consider a trust domain $(\mathcal{T}, \preceq, \sqsubseteq)$, then these characterisations are excluding a number of elements of $\mathcal{T}$ producing a subset of acceptable trust values in terms of certainty and trust respectively. So, if we define $T_\sigma, T_\tau \subseteq \mathcal{T}$ to be the set of acceptable trust values in terms of certainty and trust respectively, then the characterisations dictate that the new trust value must belong to the intersection of these sets, $T_{new} \in T_\sigma \cap T_\tau$. Or in other words, they determine the direction we should move on the trust ordering lattice or the certainty ordering c.p.o. (refer back to section 2.1) to find our new trust value T_{new}.

In the case of recommendations the calculation of their attraction is straightforward because recommendations are in fact trust values. As a result, we can evaluate a recommendation, *Rec*, by taking advantage of the structure of the trust domain $(\mathcal{T}, \preceq, \sqsubseteq)$ and directly comparing it to the current trust value in terms of certainty, $\sqsubseteq$, and trust, $\preceq$. This comparison will determine the direction of its attraction as follows:

- In terms of certainty, we calculate the greatest lower bound (glb) of T_{curr} and *Rec*. Note that because $(T, \sqsubseteq)$ is a complete partial order with a least element, the glb of any two trust values $t_1, t_2 \in T$ is guaranteed to exist. Then, there are three cases:
  1. If the $glb(T_{curr}, Rec) = T_{curr}$, then the attraction of the evidence is reinforcing.
  2. If the $glb(T_{curr}, Rec) = Rec$, then the attraction of the evidence is still reinforcing, but in this case *Rec* does not really add anything to our current trust value and can be safely ignored.
  3. Otherwise, the attraction of the evidence is contradicting.

  In both the first and the third of these cases the new trust value T_{new} must have the properties: $glb(T_{curr}, Rec) \sqsubseteq T_{new}$ and $T_{new} \sqsubseteq T_{curr}$.
- In terms of trust, T_{curr} and *Rec* are either comparable or incomparable.
  1. If they are comparable, then:
     - If $T_{curr} \preceq Rec$, then the attraction of the evidence is positive.
     - If $Rec \preceq T_{curr}$, then the attraction of the evidence is negative.

     Note that in this case the new trust value T_{new} must be inside the interval $[T_{curr}, Rec]$ or $[Rec, T_{curr}]$ respectively.
  2. If they are not comparable, then instead of comparing T_{curr} and *Rec* we compare T_{curr} to either the glb or the least upper bound (lub) of T_{curr} and *Rec*. Note that because of the fact that the $(T, \preceq)$ is a complete lattice, both the glb and the lub of any two trust values $t_1, t_2 \in T$ are guaranteed to exist. The choice between the glb and lub is a dispositional characteristic of the principals. According to this characteristic principals are classified as either *trusting*, those selecting the lub, or *distrusting*, those selecting the glb. Then there are the following two cases:
     (a) If the glb was selected, then the attraction of the evidence is negative and the new trust value T_{new} must be inside the interval defined by the glb and the current trust value T_{curr}, $T_{new} \in [glb(T_{curr}, Rec), T_{curr}]$.
     (b) If the lub was selected, then the attraction of the evidence is positive and the new trust value T_{new} must be inside the interval defined by the current trust value T_{curr} and the lub, $T_{new} \in [T_{curr}, lub(T_{curr}, Rec)]$.

The above determines the direction of the attraction, but it does not determine the *value of attraction*, $\|attr\|$. Our trust model does not provide us with a measure of distance between trust values, and as a result cannot be used for determining the value of the attraction. For this reason, we use the risk domain in order to determine the value of the attraction. For this purpose we define the notion of *risk profile distance.* Assuming risk profiles R_1 and R_2 we define their distance as:

$$diff(R_1, R_2) = \sum_{\forall o \in \mathbb{O}} \mid \mathrm{Pr}_1(x) - \mathrm{Pr}_2(x) \mid \tag{5}$$

where $\mathrm{Pr}_1(x)$ and $\mathrm{Pr}_2(x)$ are the probabilities of outcome o according to risk profile R_1 and R_2 respectively, and $\mathbb{O}$ is the set of possible outcomes. In this case, the value of a recommendation's *Rec* attraction, must be proportionate to the distance of the risk profiles associated to T_{curr} and *Rec*:

$$\|attr\| \propto diff(R_{T_{curr}}, R_{Rec}) \tag{6}$$

The exact function for the calculation of the value is up to the application developer to define.

In the case of observations, which take the form of an observed outcome, the calculation of their attraction is a bit more complicated. This calculation in general can take the following two forms:

1. *Direct Evaluation.* In this case the attraction of an observation is characterised as positive if the observed outcome incurred a benefit and negative otherwise. Further, it is characterised as reinforcing if the likelihood of the observation, *Obs*, according to the risk profile of the current trust value, $\mathrm{Pr}_{R_{T_{curr}}}(Obs) \geq 50\%$. Otherwise, it is contradicting. Its value should be proportionate to the distance of the likelihood of the observation according to the risk profile of the current trust value from 50%,

$$\|attr\| \propto \mid \mathrm{Pr}_{R_{T_{curr}}}(Obs) - 50\% \mid \tag{7}$$

   This form of observation evaluation is demonstrated in the e-purse scenario (see section 3).
2. *Indirect Evaluation.* In this case, we first produce an *evidential trust value*, T_{evd}, from the observation. Then we evaluate the attraction of T_{evd} following the approach described above in the evaluation of recommendations[2]. More specifically, we choose as T_{evd}, the trust value, T_i that is associated to the risk profile, R_i, in which the observation, *Obs* has the highest likelihood:

$$T_{evd} = T_{\{R_{max} \mid \mathrm{Pr}_{R_{max}}(Obs) = \max\limits_{i}(\mathrm{Pr}_{R_i}(Obs))\}} \tag{8}$$

   If we follow this approach, then it be should clear that considering a single observation means that T_{evd} does not offer significant insight into the trustworthiness of the principal in question. Therefore, it seems reasonable to

[2] Note that we can take the view that the recommendation adjustment does in fact produce an evidential trust value, which is subsequently used in the evaluation process.

collect a number of observations before the evaluation takes place. The collected observations construct a profile of observed behaviour. In this profile, the number of occurrences of each particular observation over the total number of observations under evaluation determines its likelihood. This profile of observed behaviour can be compared to the various profiles of expected behaviour, i.e. the risk profiles. The trust value associated to the risk profile closest to the observed one, according to the risk profile distance (see equation 5), is the evidential trust value.

The final stage in the processing of evidence is to update the current trust value T_{curr} to a new trust value T_{new} according to the attraction of the evidence. This process is carried out by an *evolve*() function. Following a similar approach to the one described in [18], there are two alternative definitions for such a function:

1. As a trust evolution function that considers a sequence of attractions in order to produce a trust value. More precisely, assuming that *AttrSeq* represents sequences of attractions:

$$evolve : AttrSeq \rightarrow \mathcal{T} \tag{9}$$

   Considering sequences of attractions instead of sets allows us to define trust evolution functions that can distinguish the past and have fixed memory. For example, we can introduce time discounting of evidence, so that more recent evidence counts for more, or we can drop evidence if it is considered to distant in the past.
2. As a trust update function that considers the current trust value and an attraction in order to produce a new trust value. More precisely, assuming that *Attr* is the set of attraction values:

$$evolve : \mathcal{T} \times Attr \rightarrow \mathcal{T} \tag{10}$$

   Trust update functions have infinite memory, since all past evidence is reflected in the trust values. This is the main reason why in the e-purse application scenarios in section 3 we use a trust update function. In fact, in this scenario the *evaluate*() and the *evolve*() function have been merged into a single function that provided with an observation, directly produces the new trust value from the current one. Note also that for any trust update function we can generate a trust evolution function by iteration starting from each initial trust value.

The exact nature of either type of *evolve*() function is up to the application developers to define. When defining the exact function they should consider the work of Jonker and Treur [18], who analyse trust evolution and update functions and identify a number of properties that such functions may have. More importantly, each of the identified properties allows the modelling of alternative principal attitudes towards trust. This type of modelling can also be applied to our trust formation and evolution approach.

Following a similar approach to [18], in our collaboration model we identify two aspects that allow us to characterise the many types of principal attitudes towards trust. We refer to these aspects as the *dispositional characteristics* of a principal. These characteristics are:

1. *Trusting Disposition.* In terms of trusting disposition, principals are classified as either *generally trusting* or *generally distrusting.* This is reflected in the initial trust value that they use for the formation process and the selection of the glb or lub in the case of incomparable trust value during evidence evaluation (see the section on recommendation evaluation above). With respect to the former, a generally trusting principal would select an initial trust value T that conveys more trust than "unknown", $\perp \preceq T$, while a generally distrusting principal one that conveys less trust than "unknown", $T \preceq \perp$. With respect to the latter, a generally trusting principal would select the lub, while a generally distrusting principal the glb.
2. *Type of Trust Dynamics.* The types of trust dynamics reflect how easily a particular principal's trust in others builds and erodes in the light of evidence. In general, principals may build or erode trust either quickly, slowly, or in balance. This means that a principal that quickly (slowly) builds trust would require a small (large) number of positive evidence to consider another principal as highly trusted. Further, a principal that quickly (slowly) erodes trust would require a small (large) number of negative evidence to consider another principal as highly distrusted, meaning that it is quite unforgiving (forgiving) of bad behaviour. Jonker and Treur suggest in [18] such a model of different types of trust dynamics. Moreover, they identify the following types:
   - Blindly Positive. A principal that after a number of good experiences with another principal will always consider it trustworthy.
   - Blindly Negative. A principal that after a number of bad experiences with another principal will always consider it untrustworthy.
   - Slow Positive, Fast Negative. A principal that requires a large number of good experiences to build trust and a small number of bad experiences to erode trust.
   - Fast Positive, Slow Negative. A principal that requires a small number of good experiences to build trust and a large number of bad experiences to erode trust.
   - Balanced Slow. A principal that requires both a large number of good experiences to build trust and a large number of bad experiences to erode trust.
   - Balanced Fast. A principal that requires both a small number of good experiences to build trust and a small number of bad experiences to erode trust.

In order to enable the expression of the above dispositional characteristics in our collaboration model, we introduce *dispositional parameters* in the *evolve*() and *evaluate*() functions. The exact nature of these parameters will of course depend on the exact definition of these functions and the characteristics of the particular application scenario. Therefore, it is up to application developers to define them. The use of dispositional parameters is demonstrated in the e-purse application scenario with the use of α_p, α_n and β that determine how slow or fast trust builds, erodes and becomes certain respectively.

2.4 Trust Formation

Although, in the previous section we examined in detail how trust evolves in the light of evidence about principals' behaviour, a complete approach to the management of trust lifecycle needs also address how trust is initially formed. In other words, we have seen that the processing of evidence and the evolution of trust values is with respect to the current trust value, which poses the question how can an initial trust value for a particular principal be derived. We refer to the process of deriving the initial trust value for a principal as *trust formation.*

The trust model (see section 2.1) has two very important properties that the trust formation process can utilise:

1. The ability to include references in the local trust policies of principals. These references enable principals to specify trust values as relations over the trust values of other principals.
2. The requirement to define over the trust domain $\mathcal{T}$ an ordering relation $\sqsubseteq$, according to which the trust values of $\mathcal{T}$ form a complete partial order with a least element $\perp$ ("unknown"). This least element represents the case where there is complete uncertainty about the behaviour of an principal. Its inclusion in the trust domain means that we have to consider and explicitly evaluate what are the risks that an interaction with an unknown principal entail.

From a trust formation point of view, the first property allows any principal to rely on others using their opinions through references. For as long as a principal does not have adequate evidence to have an opinion of its own about the trustworthiness of another principal it can remain reliant on others. However, there are no guarantees that the referenced principals would have an opinion about the trustworthiness of the principal in question either. Moreover, even if they have an opinion there are no guarantees that the principal would be able to communicate with them to find out what their opinion is. To make matters worse, there is always the case that a whole chain of references may have to be followed in order to discover what this opinion is, thus increasing the chances of a communication failure along the way. In these situations the second property becomes particularly important. It provides us with a particular trust value, namely "unknown" $\perp$, that we can always resort to.

In addition to or instead of relying on the opinions of a particular set of principals, as it would be the case when using references, we could also initiate a search for principals that have an opinion about the trustworthiness of the principal in question. In other words we can initiate a process of gathering recommendations about the principal. However, having collected a number of recommendations about the principal, the problem remains on how these recommendations can be combined in order to form an opinion about its trustworthiness.

The fact that our trust always includes the “unknown” trust value means that we always have a current trust value for all principals, even previously unknown ones. This allows us to take the view that the trust formation process is in fact just a special case of trust evolution one. This means that we can process the collected recommendations about unknown principals in exactly the same way as recommendations about known ones with the only difference being that we use $\perp$ (“unknown”) as the current trust value (see section 2.3).

The proactive collection of recommendations for previously unknown principal as part of the trust formation process, raises the issue of how these recommendations can be gathered. There are a number of approaches we could follow:

- We could bootstrap principals with a list of similar entities that could be likely sources of recommendations. However, it might be the case that none of these entities can be contacted at a particular point in time.
- We could ask unknown principals to provide us with a list of entities that could give us recommendations about them. However, this might be misleading, since no principal would recommend anyone they had a bad interaction with.
- We could ask our neighbours to suggest good recommenders. Note that the neighbourhood might have different meanings. For example, it may be comprised of our acquaintances or directly accessible entities in an ad-hoc network, etc. However, there are no guarantees that our neighbourhood is a good source of recommendations.
- We could also introduce brokers that can suggest recommenders for various types of interactions. In this case, these brokers play the role of trusted third parties.
- In some situations, however, when trustworthy recommenders are not known, a broadcasted request for recommendations might be our only option. This approach offers no guarantees whatsoever and it would probably be our last resort.

Since each one of the above approaches has its advantages and disadvantages, we may need to employ a number of them at the same time in order to form a more accurate picture about the trustworthiness of the principal in question. In general, we should be aware that the gathering of recommendations and is a tricky issue regarding careful consideration (see section 4.3 in [32] for a discussion on the tradeoffs involved in this issue).

3 E-Purse Application Scenario

Having presented our approach to the management of trust lifecycle in the previous section, in this section we show how our approach can be applied in an e-purse application scenario. We should point out that this is just an example of the application of our approach and as we have already pointed out does not cover all the various aspects. However, we believe that it is useful in demonstrating how the abstract concepts of the approach can be applied in a concrete case.

The scenario involves the use of an e-purse when a passenger is interacting with a bus company. The purpose of the e-purse is to hold a relatively small amount of e-cash (in this scenario the e-purse is limited to 100 euro) that the owner can use as if it were real cash for buying bus tickets (see figure 1).

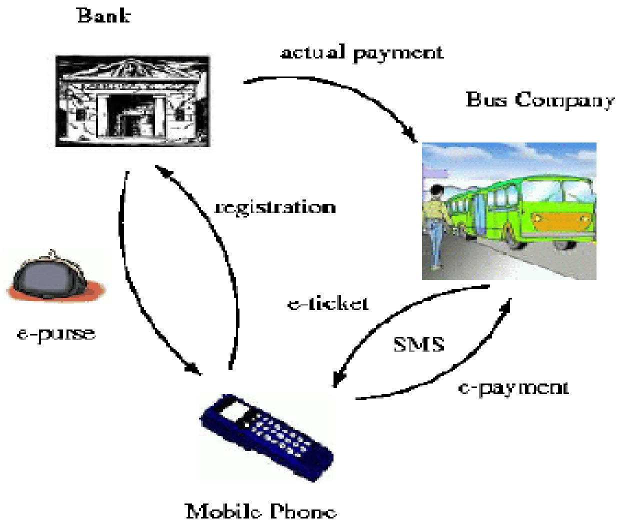


Fig. 1. E-purse scenario interaction

Users can refill their e-purse by contacting their bank provided that there is enough cash in their account. There are three different principals involved in this scenario: the passenger (owner) of the e-purse, the bus company and the bank. In this scenario we are only interested in modelling the trust relationship between the bus company and the passenger. We consider the example interaction where passengers want to purchase tickets using their e-purse.

E-cash is based on a protocol that although it protects user anonymity during normal transactions, enables identification of guilty parties in fraudulent transactions. Every time the bus company accepts e-cash in a transaction it takes the risk of losing money due to fraud. Therefore, for the bus company to decide how to respond to a purchasing request, it needs to determine the trustworthiness of the passenger. Principals can assign different levels of trust to different entities based on the available information so as to evaluate the level of risk transactions involving the user entail.

3.1 Trust Values

In the e-purse scenario the range of basic trust values is [0, 100] reflecting the amount of e-cash that the bus company is willing to accept from the requesting user. Following the interval construction technique from [7], we construct our trust values as intervals $[d_1, d_2]$ of the range of basic trust values. An interval $[d_1, d_2]$ indicates that the bus company is quite certain about the validity of amounts up to d_1 of e-cash , fairly uncertain about the validity of amounts between d_1 and d_2 and fairly certain that any amount above d_2 will be invalid. So, for any ticket purchasing request of more than d_2 the user has to pay in cash. It should be clear that the use of such trust values really simplifies the decision making process.

3.2 Risk Analysis and Decision Making

The essential step in trust exploitation is to determine expected behaviour on the basis of trust intervals. This is achieved by using the trust interval to determine the risk of interacting with a particular principal. The assumption is that the passenger's trustworthiness reflects the expected loss or gain during a transaction involving him or her. The costs involved in an interaction range from -100 to 100, denoting the maximum gain or loss for the bus company.

In the general case, the calculated risk allows entities to decide whether or not to proceed with an interaction. In this scenario, a simplified view is taken, whereby the trust value directly determines the amount of e-cash a bus company is willing to accept. The decision making process for a ticket of value x regarding a passenger with trust value $[d_1, d_2]$ is as follows:

- If $x < d_1$ then the whole amount of the transaction can be paid in e-cash.
- If $x > d_2$ then the option of paying in e-cash is not available and the full amount has to be paid in cash.
- If $d_1 < x < d_2$ then the likelihoods of the possible outcomes are examined. Note that there are only two possible outcomes, the e-cash provided by the passenger will be either valid or invalid. For the calculation of the likelihoods, we divide the range from d_1 to d_2 into a number of units, n. For example n could be equal to the price of the cheapest ticket, say 5 euro. In this case, the number of units is determined by dividing the whole range over five $(d_1 - d_2)/5$. The likelihood of invalid e-cash for each unit is (m/n), where $m = 0, 1, .., n$ (see figure 2). Note that the likelihood of invalid e-cash increases from d_1 (with a probability of 0 for invalid e-cash) to d_2 (with a probability of 1 for invalid e-cash). Considering these likelihoods for the possible outcomes the bus company can place a threshold of acceptable risk. So, it will only accept e-cash for transaction with risk below the threshold.

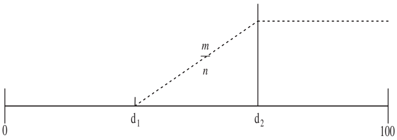


Fig. 2. Risk Analysis

3.3 Trust Evolution

In this scenario we again only consider observations and we combine the two processes of trust evolution, namely evidence evaluation and trust evolve. As a result, the attraction of every observed outcome, whether the provided e-cash were valid or not, raises or drops the boundaries of the current trust value T_{curr}.

In accordance with section 2.3, if the e-cash was valid then the attraction of the observation is considered positive, in which case we expect the lower and upper bound of T_{curr} to either remain unchanged or to be raised. While, if the e-cash was invalid the attraction of the observation is considered negative, in which case we expect the lower and upper bound of T_{curr} to either remain unchanged or to be dropped. Moreover, if the outcome was expected, i.e. its likelihood was more than 50%, then the attraction of the observation is considered reinforcing, otherwise is considered contradicting. Let us assume that m_1 and m_2 are the movements of the lower and upper bound of T_{curr} respectively. Then, in the case of reinforcing attraction we expect the size of the interval $(d_2 - d_1)$ to remain unchanged or be reduced, i.e. $m_1 > m_2$. While, in the case of contradicting attraction we expect the size of the interval to remain unchanged or be increased, i.e. $m_1 < m_2$. This is summarised in Table 1.

Table 1. Summary of observation evaluation

attraction direction	direction of boundary movement	interval size
positive, reinforcing	$\longrightarrow$	$m_1 > m_2$
positive, contradicting	$\longrightarrow$	$m_1 < m_2$
negative, reinforcing	$\longleftarrow$	$m_1 > m_2$
negative, contradicting	$\longleftarrow$	$m_1 < m_2$

For example, let us assume that t denotes the amount of e-cash in the observed transaction and $T_{curr} = [d_1, d_2]$. Then, the movements m_1 and m_2 of the lower and upper bound of T_{curr} could be determined as follows:

1. If $t < d_1$, then
   - If the e-cash is valid, then the attraction is reinforcing and positive. In this case, the observation does not really contribute any additional information about the principal and is therefore ignored, i.e.:

$$m_1 = 0 \text{ and } m_2 = 0 \tag{11}$$

   Note that if we do not ignore this kind of observations, but instead use them to raise the trustworthiness of the principal, then we are exposing ourselves to the typical trust exploitation scam, where a large number of very small value transactions could allow a transaction of a substantial value to take place even though there is no evidence to support this decision.

- If the e-cash is invalid, then the attraction is contradicting and negative. In this case:

$$m_1 = \alpha_n \times (t - d_1) \text{ and } m_2 = \beta \times m_1 \tag{12}$$

2. If $t > d_1$, then
   - If the e-cash is valid and the likelihood of t being valid is less than 50%, $Pro(t, valid) < 50\%$, then the attraction is positive and contradicting. In this case:

$$m_1 = \beta \times m_2 \text{ and } m_2 = \alpha_p \times ((100 - d_2)/(d_2 - t)) \times (1 - Pro(t, valid)) \tag{13}$$

   - If the e-cash is valid and the likelihood of t being valid is greater than 50%, $Pro(t, valid) > 50\%$, then the attraction is positive and reinforcing. In this case:

$$m_1 = \alpha_p \times (t - d_1) \times (1 - Pro(t, valid)) \text{ and } m_2 = 0 \tag{14}$$

   - If the e-cash is invalid and $Pro(t, invalid) > 50\%$, then the attraction is negative and reinforcing. In this case:

$$m_1 = 0 \text{ and } m_2 = \alpha_n \times (t - d_2) \times (1 - Pro(t, invalid)) \tag{15}$$

   - If the e-cash is invalid and $Pro(t, invalid) < 50\%$, then the attraction is negative and contradicting. In this case:

$$m_1 = \alpha_n \times (d_1/(d_1 - t)) \times (1 - Pro(t, invalid)) \text{ and } m_2 = \beta \times m_1 \tag{16}$$

Note that α_p, α_n and β range from [0, 1] and are dispositional parameters that determine how slow or fast are the positive, negative and certainty dynamics respectively. If $\alpha_p > 0.5$ then we are talking about fast positive dynamics, while if $\alpha_p < 0.5$ we are talking about slow positive dynamics. Similarly, depending on the value of α_n we are talking about fast negative or slow negative dynamics. Moreover, if $\alpha_p = \alpha_n$ then we are talking about balanced slow or fast dynamics (see section 2.3). At the same time, if β is small we reduce the size of the trust interval quickly, while if β is large we reduce it slowly.

A Specific Example. Suppose that a passenger with a trust value [20, 70] paid valid e-cash worth 40 euro to the bus company. Supposing that the range between 20 and 70 is divided into 5 units each with a size of 10, the likelihoods of the two outcomes, valid or invalid e-cash, are: 20% for invalid and 80% for valid. So, the attraction of this observation is positive and reinforcing. Applying the functions described above using $\alpha_p = 0.5$, we have $m_1 = 2$, $m_2 = 0$. So, the new trust value $T_{new} = [d_1 + m_1, d_2 + m_2] = [20 + 2, 70 + 0] = [22, 70]$.

4 Comparison to the State of the Art

Part of the motivation for this was work was the weaknesses of certificate-based approaches to trust management [4, 5, 20, 22, 28] in terms of how trust is formed and evolves over time. These weaknesses deemed these approach inadequate for a global computing environment. It should clear from the above presentation that the SECURE approach to trust lifecycle management with its evidence-based approach and its emphasis on evidence processing and trust formation addresses these weaknesses. However, in recent years there have been other attempts at more intuitive computational models of trust, with a basis in the history of past interactions along similar lines to the SECURE approach [1, 11, 12, 17, 18, 19, 21, 23, 27, 31, 24, 36, 38]. These attempts also improve on the certificate-based approaches and have introduced particular concepts that we have also found useful and have incorporated into our model, for example:

- The explicit modelling of risk [12, 17].
- The explicit modelling of uncertainty [19, 38].
- The notion of trust in the recommender and recommendation adjustment [1, 19, 21, 36]
- The use of trust update and trust evolution function for the incorporation of evidence to the current trust value [18].
- The various types of trust dynamics [18].

Besides the fact that the SECURE approach is a unique combination of the above concepts, it has also the following discriminating features:

- It emphasises principal recognition over authentication.
- It defines a clear relationship between trust and risk.
- It introduces the notion of attraction, which guides the processing of evidence both in terms of trustworthiness and certainty.

These characteristics are what makes the SECURE approach to trust lifecycle management more suitable for a global computing environment.

5 Conclusions and Future Work

In conclusion, the goal in a global computing environment is to enable entities to collaborate by allowing them to make autonomous access control decisions with partial information about their potential collaborators. The SECURE project aims to achieve this goal by using trust as the mechanism for managing the risks and the uncertainty that collaborations in such an environment entail. Core in this mechanism is the ability to manage the trust lifecycle as a procedure of collecting and processing evidence. For this purpose a preliminary collaboration model has been defined which addresses both theoretical and operational issues of trust lifecycle management, namely trust formation, trust evolution and trust exploitation. In this paper we have only focused on the theoretical issues of the collaboration model. For a presentation of the operational issues we point the interested reader to section 4 of [32]. At the heart of evidence processing according to the collaboration model is the notion of attraction, which considers the effects of evidence about the behaviour of a particular principal on its current trust value both in terms of trustworthiness and certainty. The notion of attraction along with the emphasis on principal recognition and the definition of a clear relationship between trust and risk is what discriminates the SECURE project from other work on evidence-based trust management and makes it more appropriate for a global computing setting.

The preliminary collaboration model has been instrumental in the SECURE project in driving the developments for both the trust and the risk model as well as the SECURE framework architecture and kernel implementation. It has highlighted a number of issues particularly regarding the adopted relationship between trust and risk, which refinements of the underlying trust and risk models are currently trying to address. For the developments on this front we point the interested reader to [8] and [2] respectively.

We are currently in the process of evaluating our approach to evidence processing. The evaluation is along two dimensions. First, we examine the applicability of our model on a number of different application scenarios from [6]. We have already applied our model to a Smart Space application scenario (see section 6.1 in [32]). Second, we are developing a simulation framework, which will allow us to experiments with a variety of alternative ways of processing evidence, i.e. the various functions that the model leaves for the application developer to define. We have already developed a simulator for the e-purse application scenario presented above. The initial results of the simulation are encouraging and we are planning to report our conclusions in more detail in the near future.

Finally, regarding the development of the collaboration model itself our main goal for the near future is to introduce a notion of context to the model. Context aims to capture the situational character of trust, i.e. the fact that the trustworthiness of a principal varies in different situations. For this reason we view context as situational modifier of principals' behaviour. Some preliminary considerations about context have been already presented in section 3.2 of [32], however the whole issue certainly requires further investigation.

References

1. A. Abdul-Rahman and S. Hailes. Supporting trust in virtual communities. In *Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 6*, page 6007. IEEE Computer Society Press, January 2000.
2. Jean Bacon, Andras Belokosztolszki, Nathan Dimmock, David Eyers, David Ingram, and Ken Moody. Preliminary definition of a trust-based access control model. *SECURE Deliverable 3.2*, 2003.
3. Jean Bacon, Nathan Dimmock, David Ingram, Ken Moody, Brian Shand, and Andy Twigg. Definition of risk model. *SECURE Deliverable 3.1*, 2002.
4. Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. Keynote: Trust management for public-key infrastructures. In *Secure Internet Programming: Issues in Distributed and Mobile Object Systems*, volume 1550 of *LNCS*, pages 59–63. Springer-Verlag, 1998.
5. Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 164–173, Los Alamitos, USA, May 1996. AT&T.

6. C. Bryce, V. Cahill, G. Di Marzo Serugendo, C. English, S. Farrell, E. Gray, C. D. Jensen, P. Nixon, J.-M. Seigneur, S. Terzis, W. Wagealla, and C. Yong. Application scenarios. *SECURE Deliverable 5.1*, 2002.
7. Marco Carbone, Oliver Danvy, Ivan Damgaard, Karl Krukow, Anders Mller, Jesper B. Nielsen, and Mogens Nielsen. A model for trust. *SECURE Deliverable 1.1*, 2002.
8. Marco Carbone, Karl Krukow, and Mogens Nielsen. Revised computational trust model. *SECURE Deliverable 1.3*, 2004.
9. Marco Carbone, Mogens Nielsen, and Vladimiro Sassone. A formal model for trust in dynamic networks. In *Proceedings of the International Conference on Software Engineering and Formal Methods*, pages 54–63, Brisbane, Australia, September 2003.
10. Marco Carbone, Mogens Nielsen, and Vladimiro Sassone. A formal model for trust in dynamic networks. RS RS-03-4, BRICS, DAIMI, January 2003. 18 pp.
11. Rita Chen and William Yeager. Poblano - a distributed trust model for peer-to-peer networks. Technical report, Sun Microsystems, 2001.
12. Theo Dimitrakos. System models, e-risks and e-trust. towards bridging the gap? In *Proceedings of the 1st IFIP Conference on e-Commerce, e-Business, e-Government.* Kluwer Academic Publishers, October 2001.
13. Nathan Dimmock. How much is 'enough'? Risk in trust-based access control. In *Proceedings of the IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises: Enterprise Security (Special Session on Trust Management)*, Linz, Austria, June 2003.
14. Colin English, Sotirios Terzis, and Waleed Wagealla. Engineering trust-based collaborations in a global computing environment. In Stefan Poslad Christian Jensen and Theo Dimitrakos, editors, *Proceedings of the Second International Conference on Trust Management*, volume 2995 of *LNCS*, pages 120–134, Oxford, UK, March 2004. Springer.
15. Colin English, Waleed Wagealla, Paddy Nixon, Sotirios Terzis, Andrew McGettrick, and Helen Lowe. Trusting collaboration in global computing. In Paddy Nixon and Sotirios Terzis, editors, *Proceedings of the First International Conference on Trust Management*, volume 2692 of *LNCS*, pages 136–149, Heraklion, Crete, Greece, May 2003. Springer.
16. Diego Gambetta. Can we trust trust? In Diego Gambetta, editor, *Trust: Making and Breaking Cooperative Relations*, pages 213–237, Oxford, 1990. Basil Blackwell.
17. Tyrone Grandison and Morris Sloman. Specifying and analysing trust for internet applications. In *Proceeedings of the 2nd IFIP IEEE Conference*, pages 145–157, October 2002.
18. Catholijn M. Jonker and Jan Treur. Formal analysis of models for the dynamics of trust based on experiences. In Francisco J. Garijo and Magnus Boman, editors, *Proceedings of the 9th European Workshop on Modelling Autonomous Agents in a Multi-Agent World : Multi-Agent System Engineering (MAAMAW-99)*, volume 1647 of *LNCS*, pages 221–231. Springer-Verlag, June 1999.
19. Audun Jøsang. A logic for uncertain probabilities. *Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 9(3):279–311, June 2001.
20. Lalana Kagal, Jeffrey L Undercoffer, Filip Perich, Anupam Joshi, and Tim Finin. A security architecture based on trust management for pervasive computing systems. In *Grace Hopper Celebration of Women in Computing*, October 2002.

21. Michael Kinateder and Kurt Rothermel. Architecture and algorithms for a distributed reputation system. In Paddy Nixon and Sotirios Terzis, editors, *Proceedings of the First International Conference on Trust Management*, volume 2692 of *LNCS*, pages 1–16, Heraklion, Crete, Greece, May 2003. Springer.
22. Ninghui Li and John C. Mitchell. RT: A role based trust management framework. In *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition (DISCEX III)*. IEEE Computer Society Press, April 2003.
23. Stephen Marsh. *Formalising Trust as a Computational Concept*. PhD thesis, University of Stirling, 1994.
24. M. Marx and J. Treur. Trust dynamics formalised in temporal logic. In L. Chen and Y. Zhuo, editors, *Proceedings of the 3rd International Conference on Cognitive Science, ICCS*, pages 359–363. USTC Press, Beijing, 2001.
25. John McLean. Security models. In J. Marciniak, editor, *Encyclopedia of Software Engineering*. John Wiley & Sons, 1994.
26. Ravi Sandhu. Access control: The neglected frontier. In *Proceedings of the First Australian Conference on Information Security and Privacy*, volume 1172 of *LNCS*, pages 219–227, Wollong, Australia, June 1996. Springer.
27. V. Schmatikov and C. Talcott. Reputation-based trust management (extended abstract). In *Proceedings of the Workshop on Issues in the Theory of Security (WITS)*, 2003.
28. K. E. Seamons, M. Winslett, T. Yu, B. Smith, E. Child, J. Jacobson, H. Mills, and L. Yu. Requirements for policy languages for trust negotiation. In *Proceedings of the 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002)*, pages 68–79, Monterey, CA, USA, June 2002.
29. J.-M. Seigneur, S. Farrell, C. Jensen, E. Gray, and C. Yong. End-to-end trust starts with recognition. In *Proceedings of the First International Conference on Security in Pervasive Computing*, 2003.
30. B. Shand, N. Dimmock, and J. Bacon. Trust for Ubiquitous, Transparent Collaboration. In *Proceedings of the First IEEE Annual Conference on Pervasive Computing and Communications (PerCom 2003)*, pages 153–160, Dallas-Ft. Worth, TX, USA, March 2003.
31. Yao-Hua Tan and Walter Thoen. Formal aspects of a generic model of trust for electronic commerce. In *Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 6*, page 6006. IEEE Computer Society Press, January 2000.
32. Sotirios Terzis, Waleed Wagealla, Colin English, and Paddy Nixon. The secure collaboration model. Technical Report Smartlab-03-2003, Dept. of Computer and Information Sciences, University of Strathclyde, December 2003.
33. Global Computing Initiative Website. http://www.cordis.lu/ist/fet/gc.htm, 2002.
34. Kerberos Website. http://web.mit.edu/kerberos/www/.
35. SECURE Project Official Website. http://secure.dsg.cs.tcd.ie, 2002.
36. Li Xiong and Ling Liu. Building trust in decentralized peer-to-peer electronic communities. In *Proceedings of the 5th International Conference on Electronic Commerce Research(ICECR-5)*, Montreal, Canada, October 2002.
37. Li Xiong and Ling Liu. A reputation-based trust model for peer-to-peer ecommerce communities. In *Proceedings of the 4th ACM conference on Electronic commerce*, pages 228–229, San Diego, CA, USA, 2003. ACM Press.
38. Bin Yu and Munindar P. Singh. An evidential model of distributed reputation management. In *Proceedings of the first international joint conference on Autonomous agents and multiagent systems*, pages 294–301, Bologna, Italy, 2002. ACM Press.

The SOCS Computational Logic Approach to the Specification and Verification of Agent Societies

Marco Alberti[1], Federico Chesani[2], Marco Gavanelli[1], Evelina Lamma[1],
Paola Mello[2], and Paolo Torroni[2]

[1] Dip. di Ingegneria - Università di Ferrara - Via Saragat, 1 - 44100 Ferrara, Italy
{malberti, mgavanelli, elamma}@ing.unife.it
[2] DEIS - Università di Bologna - Viale Risorgimento, 2 - 40136 Bologna, Italy
{fchesani, pmello, ptorroni}@deis.unibo.it

Abstract. This article summarises part of the work done during the first two years of the SOCS project, with respect to the task of modelling interaction amongst CL-based agents. It describes the SOCS social model: an agent interaction specification and verification framework equipped with a declarative and operational semantics, expressed in terms of abduction. The operational counterpart of the proposed framework has been implemented and integrated in SOCS-SI, a tool that can be used for on-the-fly verification of agent compliance with respect to specified protocols.

1 Introduction

Computees are Computational Logic-based entities interacting in the context of global and open computing systems [1]. They are abstractions of the entities that populate Global Computing (GC) environments [2]. These entities can form complex organizations, that we call *Societies Of ComputeeS* (SOCS, for short) [3]. The main objective of Global Computing, rephrased in terms of SOCS, is to provide a solid scientific foundation for the design of societies of computees, and to lay the groundwork for achieving effective principles for building and analyzing such systems. Between January 2002 and March 2004, the project developed a society formal model to satisfy the high-level objectives derived directly from the GC vision of an *open* and *changing* environment.

In this context, by "open" environment we mean, following Hewitt's work [4] about information systems and then Artikis et al.'s [5] about computational societies, an environment or society where the following properties hold:

(*i*) the behavior of members and their interactions are unpredictable (i.e., the evolution of the society is non-deterministic);
(*ii*) the internal architecture of each member is neither publicly known nor observable (thus, members may have heterogeneous architectures);
(*iii*) members of the society do not necessarily share common goals, desires or intentions (i.e., each member may conflict with others when trying to reach its own purposes).

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 314–339, 2004.

This definition of openness is based on externally observable features within the society. It caters for heterogeneous and possibly non-cooperative members. Therefore, our model of society will not constrain the ways computees join or leave a society, it will emphasize the presence of heterogeneous computees in the same society, and it will assume that the internal structure of computees is not guaranteed to be observable, or their social behaviour predictable.

The SOCS social model specifies a social knowledge which interprets and gives a social meaning to the members' social behavior. It supports the notion of *social goal*, allowing for both goal-directed and non-goal-directed societies.

In our approach, we believe that the knowledge and technologies acquired so far in the area of Computational Logic provide a solid ground to build upon. At the society level, the role of Computational Logic is to provide both a declarative and an operational semantics to interactions. The advantages of such an approach are to be found:

(*i*) in the design and specification of societies of computees, based on a formalism which is declarative and easily understandable by the user;
(*ii*) in the possibility to detect undesirable behavior, through *on the fly* control of the system based on the computees' observable behavior (communication exchanges) and dynamic conformance check of such behaviour with the constraints posed by the society. Interestingly, as we will see, this can be achieved by exploiting a suitable proof procedure which is the operational counterpart of the mentioned formalism;
(*iii*) in the possibility to (formally) prove properties of protocols and societies.

Therefore, in our approach, we define the (semantics of) protocols and communication languages as logic-based integrity constraints over social events (e.g., communicative acts), called Social Integrity Constraints (ic_S) [6].

The ideal "correct" behaviour of a society is modelled as expectations about events. ic_S define the expectations stemming from a certain history of events and possibly a set of goals. Expectations and ic_S are the formalism used to define the "social semantics" of agent communication languages and interaction protocol: a semantics which is verifiable without having any knowledge about the agents' internals.

The syntax of ic_S and of the society in general are those of a suitably extended logic program. In fact, we define the "social knowledge" by assimilating it to abductive logic programs [7], and we define a notion of *expected social events*, by expressing them as abducible predicates, while using ic_S to constrain the "socially admissible" communication patterns of computees (i.e., those who match the expectations).

The society infrastructure is devoted to checking the compliance of the society members' behaviour, with respect to its expectations.

The compliance check is based on a proof-procedure called $\mathcal{S}$CIFF. $\mathcal{S}$CIFF, standing for "IFF, augmented with Constraints, for handling agent $\mathcal{S}$ocieties", is an extension of the well known IFF abductive logic programming proof-procedure, defined by Fung and Kowalski [8]. The $\mathcal{S}$CIFF extends the IFF in a number of directions: it provides both a richer syntax of abductive theories (programs and integrity constraints), it caters for interactive event assimilation, it supports fulfillment check and violation detection, and it embodies CLP-like constraints [9] in the ic_S.

The $\mathcal{S}$CIFF has been proven sound with respect to the declarative semantics of the society model, in its ALP interpretation [10].

The $\mathcal{S}$CIFF has been implemented and integrated into a Java-Prolog-CHR based tool, named SOCS-SI (SOCS Social Infrastructure [11]). This implementation can be used to verify that agents comply to a Social Integrity Constraints-based specification. The intended use of SOCS-SI is in combination with agent platforms, such as PROSOCS [12], for on-the-fly verification of compliance to protocols. In SOCS-SI, $\mathcal{S}$CIFF is part of an integrated environment, provided with interface modules to allow for such a combination, and with a graphical user interface to observe the actual behaviour of the society members with respect to their expected behaviour, and to detect possible deviations.

The main innovative contribution of the SOCS social model, under a Global Computing perspective, resides in the foundational aspects of the SOCS society model and in its direct link with its implementation, SOCS-SI.

The present work is meant to survey the activity undergone within the first two years of the SOCS project, with respect to the society infrastructure, and in the context of Global Computing. For a more detailed description of specific aspects, the reader can refer to the articles cited in the bibliography.

The paper is structured as follows. In Section 2, we present the formal model for societies, and its declarative semantics. Section 3 presents the $\mathcal{S}$CIFF proof procedure. Its implementation, and the overall tool SOCS-SI is described in Section 4. We discuss related work in Section 5, and we conclude and outline future work in Section 6.

2 SOCS Social Model

The SOCS model describes the knowledge about society in a declarative way. Such knowledge is mainly composed of two parts: a *static* part, defining the society organizational and "normative" elements, and a *dynamic* part, describing the "socially relevant" events, that have so far occurred. In most of our examples, events will be communicative acts, in line with most work done on software agents. However, this is not necessarily the case. Depending on the context in which this model is instantiated, socially relevant events could indeed be physical actions or transactions, such as electronic payments. In addition to these two categories of knowledge, information about social *goals* is also maintained.

In our model, the society is time by time aware of social events that dynamically happen in the social environment (*happened* events). The "normative elements" are encoded in what we call ic_S, as we will show below. Based on the available history of events, on its specification of ic_S and its goals, the society can define what the "expected social events" are and what the social events that are expected *not* to happen. The expected events, from a normative perspective, reflect the "ideal" behaviour of the computees. We call these events *social expectations*.

2.1 Representation of the Society Knowledge

The knowledge in a society S is given by the following components:

- a (static) *Social Organization Knowledge Base*, denoted $SOKB$;
- a (static) set of *Social Integrity Constraints* (IC_S), denoted $\mathcal{IC}_S$; and
- a set of *Goals* of the society, denoted by $\mathcal{G}$.

In the following, the terms *Atom* and *Literal* have the usual Logic Programming meaning [13].

A society may evolve, as new events happen, giving rise to sequence of society instances, each one characterised by the previous knowledge components and, in addition, a (dynamic) *Social Environment Knowledge Base*, denoted $SEKB$.

In particular, $SEKB$ is composed of:

- *Happened events*: atoms indicated with functor **H**;
- *Expectations*: events that should (but might not) happen (atoms indicated with functor **E**), and events that should not (but might indeed) happen (atoms indicated with functor **EN**).

In our context, "happened" events are not all the events that have actually happened, but only those observable from the outside of agents, and relevant to the society. The collection of such events is the history, **HAP**, of a society instance. Events are represented as ground atoms

$$\mathbf{H}(\mathit{Event}[, \mathit{Time}]).$$

Expectations can be

$$\mathbf{E}(\mathit{Event}[, \mathit{Time}]) \qquad \mathbf{EN}(\mathit{Event}[, \mathit{Time}])$$

for, respectively, positive and negative expectations. **E** is a positive expectation about an event (the society expects the event to happen) and **EN** is a negative expectation, (the society expects the event not to happen[1]). Explicit negation ($\neg$) can be applied to expectations.

The arguments of expectation atoms can be non-ground terms. Intuitively, if an $\mathbf{E}(X)$ atom is in the set of expectations generated by the society, $\mathbf{E}(X) \in \mathbf{EXP}$, "**E**" indicates a wish about an event $\mathbf{H}(Y) \in \mathbf{HAP}$ which unifies with it: X/Y. One such event will be enough to fulfill the expectation: thus, variables in an **E** atom are always existentially quantified.

For instance, in an auction context such as the one exemplified in [14], the following atom:

$$\mathbf{E}(\mathit{tell}(\mathit{Auctioneer}, \mathit{Bidders}, \mathit{openauction}(\mathit{Item}, \mathit{Dialogue})), T_{open})$$

could stand for an expectation about a communicative act *tell* made by a computee (*Auctioneer*), addressed to a (group of) computees (*Bidders*), with subject *openauction*($\mathit{Item}, \mathit{Dialogue}$), at a time T_{open}.

[1] **EN** is a shorthand for **E** *not*.

The following scope rules and quantifications are adopted:

- variables in **E** atoms are always existentially quantified with scope the entire set of expectations
- the other variables, that occur only in **EN** atoms are universally quantified (the scope of universally quantified variables is not important, as $\forall X.p(X) \wedge q(X)$ is logically equivalent to $\forall X.p(X) \wedge \forall Y.q(Y)$).

The *SOKB* defines structure and properties of the society, namely: goals, roles, and common knowledge and capabilities. *SOKB* can change from time to time. However, this knowledge can be seen as *static* since it describes the organization of a society which changes more slowly than the way the *SEKB* does. The SOKB is a logic program, consisting of clauses

$$\begin{aligned} Clause &::= Atom \leftarrow Body \\ Body &::= ExtLiteral\ [\ \wedge ExtLiteral\ ]^{\star} \\ ExtLiteral &::= Literal \mid Expectation \mid Constraint \\ Expectation &::= [\neg]\mathbf{E}(Event\ [,T]) \mid [\neg]\mathbf{EN}(Event\ [,T]) \end{aligned} \tag{1}$$

In a clause, the variables are quantified as follows:

- Universally, if they occur only in literals with functor **EN** (and possibly constraints), with scope the body;
- Otherwise universally, with scope the entire *Clause*.

We call *definite* the predicates for which there exists a definition; i.e., a predicate whose name occurs in at least the head of a clause.

The following is a sample clause:

$$\begin{aligned} & sold(Item) \leftarrow \\ & \qquad \mathbf{E}(tell(Auctioneer, Bidders, openauction(Item, Dialogue)), T_{open}) \end{aligned} \tag{2}$$

It says that one way to fulfill the goal: “to have a certain *item* sold,” could be to have some computee acting as an auctioneer and telling a set of possible bidders that an auction is open for the item.

The goal $\mathcal{G}$ of the society has the same syntax as the *Body* of a clause in the SOKB, and the variables are quantified accordingly.

As an example, we can consider a society with the goal of selling items. In order to sell an item, the society might expect some computee to embody the role of auctioneer. The goal of the society could be

$$\leftarrow sold(nail)$$

and the society might have, in the *SOKB*, a rule such as Eq. 2. Indeed, there could be more clauses specifying other ways of achieving the same goal, like expecting some computee to advertise a sale on some public channel, or generating an expectation about a request of that item by some potential customer agent. The protocol of the auction (i.e., the way the auctioneer and the bidders are expected to interact, and in particular, to interact in a socially meaningful way) can be then specified by means of ic_S.

Social Integrity Constraints are in the form of implications. We report here, for better readability, the characterizing part of their syntax:

$$\begin{aligned}
ic_S &::= \chi \rightarrow \phi \\
\chi &::= (HEvent|Expectation)\ [\wedge BodyLiteral]^\star \\
BodyLiteral &::= HEvent|Expectation|Literal|Constraint \\
\phi &::= HeadDisjunct\ [\ \vee HeadDisjunct\ ]^\star|\bot \\
HeadDisjunct &::= Expectation\ [\ \wedge (Expectation|Constraint)]^\star \\
Expectation &::= [\neg]\mathbf{E}(Event\ [,T])\ |\ [\neg]\mathbf{EN}(Event\ [,T]) \\
HEvent &::= [\neg]\mathbf{H}(Event\ [,T])
\end{aligned} \tag{3}$$

Given an ic_S $\chi \rightarrow \phi$, χ is called the *body* (or the *condition*) and ϕ is called the *head* (or the *conclusion*).

The rules of scope and quantification are as follows:

1. Any variable in an ic_S must occur in at least an *Event* or in an *Expectation*.
2. The variables that occur both in the body and in the head are quantified universally with scope the entire ic_S.
3. The variables that occur only in the head (and must occur in at least one *Expectation*, by rule 1)
   (a) if they occur in literals **E** or ¬**E** are quantified existentially and have as scope the disjunct they belong to;
   (b) otherwise they are quantified universally.
4. The variables that occur only in the body are quantified inside the body as follows:
   (a) if they occur only in conjunctions of ¬**H**, **EN**, ¬**EN** or *Constraint*s are quantified universally;
   (b) otherwise are quantified existentially.
5. The order of the quantifiers is indeed meaningful. In our syntax, the quantifier ∀ cannot be followed by ∃.

The following ic_S models one of the auction rules, stating that each time a bidding event happens, the auctioneer should have sent before an *openauction* event (to all bidders).

$$\begin{aligned}
&\mathbf{H}(current\_time, T_c), \mathbf{H}(tell(S, R, bid(Item, P), A_{number}), T_{bid}), T_{bid} < T_c \\
&\rightarrow \mathbf{E}(tell(R, Bidders, openauction(Item, A_{number})), T_{open}), T_{open} \leq T_c
\end{aligned}$$

2.2 ALP Interpretation of the Society

SOCS social model has been interpreted in terms of Abductive Logic Programming [7], and an abductive semantics has been proposed for it [15]. Abduction has been widely recognised as a powerful mechanism for hypothetical reasoning in the presence of incomplete knowledge [16–19]. Incomplete knowledge is handled by labeling some pieces of information as abducibles, i.e., possible hypotheses which can be assumed, provided that they are consistent with the current knowledge base. More formally, given a theory T and a formula G, the goal of abduction is to find a (possibly minimal) set of atoms Δ which together with T "entails" G, with respect to some notion of "entailment" that the language of T is equipped with.

An Abductive Logic Program (ALP, for short) [7] is a triple $\langle KB, \mathcal{A}, IC\rangle$ where KB is a logic program, (i.e., a set of clauses), $\mathcal{A}$ is a set of predicates that are not defined in KB and that are called *abducibles*, IC is a set of formulae called *Integrity Constraints*. An abductive explanation for a goal G is a set $\Delta \subseteq \mathcal{A}$ such that $KB \cup \Delta \models G$ and $KB \cup \Delta \models IC$, for some notion of entailment $\models$.

In our social model, the idea is to exploit abduction for defining expected behaviour of the computees inhabiting the society, and an abductive proof procedure such as the $\mathcal{S}$CIFF to dynamically *generate* the expectations and perform the *compliance check*. By "compliance check" we mean the procedure of checking that the ic_S are not violated, together with the function of detecting fulfillment and violation of expectations.

Before we give the declarative semantics of the SOCS social model, we formalise better the notions of *instance* of a society, and *closure* of an instance of a society.

Definition 1. *An* instance $\mathcal{S}_{\mathbf{HAP}}$ *of a society* $\mathcal{S}$ *is represented as an ALP, i.e., a triple* $\langle P, \mathcal{E}, \mathcal{IC}_S\rangle$ *where:*

- P *is the SOKB of* $\mathcal{S}$ *together with the history of happened events* **HAP***;*
- $\mathcal{E}$ *is the set of* abducible predicates*, namely* **E***,* **EN***,* ¬**E***,* ¬**EN***;*
- $\mathcal{IC}_S$ *are the social integrity constraints of* $\mathcal{S}$*.*

The set **HAP** characterises the instance of a society, and represents the set of *observable* and *relevant* events for the society which have already happened. Note that we assume that such events are always ground.

A society instance is closed, when its characterizing history has been closed under the Closed World Assumption (CWA), i.e., when no further event might occur. In the following, we indicate a closed history by means of an overline: $\overline{\mathbf{HAP}}$.

2.3 Declarative Semantics

We give semantics to a society instance by defining those sets of expectations which, together with the society's knowledge base and the happened events, imply an instance of the goal—if any—and *satisfy* the integrity constraints.

In our definition of integrity constraint satisfaction we will rely upon a notion of entailment in a three-valued logic, being it more general and capable of dealing with both open and closed society instances. Therefore, in the following, the symbol $\models$ has to be interpreted as a notion of entailment in a three-valued setting [20].

Throughout this section, as usual when defining declarative semantics, we always consider the ground version of social knowledge base and integrity constraints, we do not consider CLP-like constraints. Moreover, we omit the time argument in events and expectations.

We first introduce the concept of *$\mathcal{IC}_S$-consistent set of social expectations*[2]. Intuitively, given a society instance, an $\mathcal{IC}_S$-consistent set of social expectations is a set of expectations about social events that are compatible with P (i.e., the $SOKB$ and the set $\mathbf{HAP}$), and with $\mathcal{IC}_S$.

Definition 2 ($\mathcal{IC}_S$-Consistency). *Given a (closed/open) society instance $\mathcal{S}_{\mathbf{HAP}}$, an $\mathcal{IC}_S$-*consistent *set of social expectations* $\mathbf{EXP}$ *is a set of expectations such that:*

$$SOKB \cup \mathbf{HAP} \cup \mathbf{EXP} \models \mathcal{IC}_S \tag{4}$$

(Notice that for closed instances $\mathbf{HAP}$ *has to be read* $\overline{\mathbf{HAP}}$*).*

In definition 2 (and in the following definitions 5, 6, 7 and 8), for open instances we refer to a three-valued completion where only the history of events has not been completed. Therefore, for open instances,

$$SOKB \cup \mathbf{HAP} \cup \mathbf{EXP} \models \mathcal{IC}_S$$

is a shorthand for:

$$Comp(SOKB \cup \mathbf{EXP}) \cup \mathbf{HAP} \cup CET \models \mathcal{IC}_S$$

where $Comp()$ is three-valued completion [20] and CET is Clark's equational theory.

For closed instances, instead,

$$SOKB \cup \overline{\mathbf{HAP}} \cup \mathbf{EXP} \models \mathcal{IC}_S$$

is a shorthand for:

$$Comp(SOKB \cup \mathbf{EXP} \cup \mathbf{HAP}) \cup CET \models \mathcal{IC}_S$$

since also the (closed) history of events needs to be completed.

$\mathcal{IC}_S$-consistent sets of expectations can be self-contradictory (e.g., both $\mathbf{E}(p)$ and $\neg\mathbf{E}(p)$ may belong to a $\mathcal{IC}_S$-consistent set). In particular, among the $\mathcal{IC}_S$-consistent sets of expectations, we are interested in those which are also consistent from the viewpoint of our intended use of expectations, i.e., in relation to the semantics of interactions. We will say that a set $\mathbf{EXP}$ is E-consistent if it does not contain both a positive and a negative expectation of the same event, and that it is $\neg$-consistent if it does not contain both an expectation and its explicit negation:

Definition 3 (E-Consistency). *A set of social expectations* $\mathbf{EXP}$ *is* E-consistent *if and only if for each (ground) term p:*

$$\{\mathbf{E}(p), \mathbf{EN}(p)\} \not\subseteq \mathbf{EXP}$$

[2] With abuse of terminology, we call this notion $\mathcal{IC}_S$-consistency though it corresponds to the theoremhood view rather than to the consistency view defined in [8].

Definition 4 ($\neg$-Consistency). *A set of social expectations* $\mathbf{EXP}$ *is* $\neg$-consistent *if and only if for each (ground) term p:*

$$\{\mathbf{E}(p), \neg\mathbf{E}(p)\} \not\subseteq \mathbf{EXP} \qquad \textit{and} \qquad \{\mathbf{EN}(p), \neg\mathbf{EN}(p)\} \not\subseteq \mathbf{EXP}.$$

Given a closed (respectively, open) society instance, a set of expectations is called *closed* (resp. *open*) *admissible* if and only if it satisfies Definitions 2, 3 and 4, i.e., if it is $\mathcal{IC}_S$-, E- and $\neg$-consistent.

Definition 5 (Fulfillment). *Given a (closed/open) society instance* $\mathcal{S}_{\mathbf{HAP}}$*, a set of social expectations* $\mathbf{EXP}$ *is* fulfilled *if and only if for all (ground) terms p:*

$$\mathbf{HAP} \cup \mathbf{EXP} \cup \{\mathbf{E}(p) \rightarrow \mathbf{H}(p)\} \cup \{\mathbf{EN}(p) \rightarrow \neg\mathbf{H}(p)\} \not\models \bot \tag{5}$$

Notice that Definition 5 requires, for a closed instance of a society, that each positive expectation in $\mathbf{EXP}$ has a corresponding happened event in $\mathbf{HAP}$, and each negative expectation in $\mathbf{EXP}$ has no corresponding happened event. This requirement is weaker for open instances, where a set $\mathbf{EXP}$ is not fulfilled only when a negative expectation occurs in the set, but the corresponding event happened (i.e., the implication $\mathbf{EN}(p) \rightarrow \neg\mathbf{H}(p)$ is false).

Symmetrically, we define violation:

Definition 6 (Violation). *Given a (closed/open) society instance* $\mathcal{S}_{\mathbf{HAP}}$*, a set of social expectations* $\mathbf{EXP}$ *is* violated *if and only if there exists a (ground) term p such that:*

$$\mathbf{HAP} \cup \mathbf{EXP} \cup \{\mathbf{E}(p) \rightarrow \mathbf{H}(p)\} \cup \{\mathbf{EN}(p) \rightarrow \neg\mathbf{H}(p)\} \models \bot \tag{6}$$

Finally, we give the notion of goal achievability and achievement.

Definition 7 (Goal Achievability). *Given an open instance of a society,* $\mathcal{S}_{\mathbf{HAP}}$*, and a ground goal G, we say that G is* achievable *(and we write* $\mathcal{S}_{\mathbf{HAP}} \mathrel{|}\!\!\approx_{\mathbf{EXP}} G$*) iff there exists an (open) admissible and fulfilled set of social expectations* $\mathbf{EXP}$*, such that:*

$$SOKB \cup \mathbf{HAP} \cup \mathbf{EXP} \models G \tag{7}$$

(which, as explained earlier, is a shorthand for $Comp(SOKB \cup \mathbf{EXP}) \cup \mathbf{HAP} \cup CET \models G$).

Definition 8 (Goal Achievement). *Given a closed instance of a society,* $\mathcal{S}_{\overline{\mathbf{HAP}}}$*, and a ground goal G, we say that G is* achieved *(and we write* $\mathcal{S}_{\overline{\mathbf{HAP}}} \models_{\mathbf{EXP}} G$*) iff there exists a (closed) admissible and fulfilled set of social expectations* $\mathbf{EXP}$*, such that:*

$$SOKB \cup \overline{\mathbf{HAP}} \cup \mathbf{EXP} \models G \tag{8}$$

(i.e., $Comp(SOKB \cup \mathbf{HAP} \cup \mathbf{EXP}) \cup CET \models G$).

3 Operational Framework

The $\mathcal{S}$CIFF proof procedure is inspired to the IFF proof procedure [8]. As the IFF, it is based on a transition system, that rewrites a formula into another, until no more rewriting transitions can be applied (quiescence). Each of the transitions generates one or more children from a node. As an extension of the IFF, the $\mathcal{S}$CIFF also has to deal with (*i*) universally quantified variables in abducibles (*ii*) dynamically incoming events (*iii*) consistency, fulfillment and violations (*iv*) CLP-like constraints.

Each node of the proof procedure is represented by the tuple

$$T \equiv \langle R, CS, PSIC, \mathbf{PEXP}, \mathbf{HAP}, \mathbf{FULF}, \mathbf{VIOL} \rangle$$

where

- R is a conjunction (that replaces the *R*esolvent in SLD resolution); initially set to the goal G, the conjuncts can be atoms or disjunctions (of conjunctions of atoms)
- CS is the constraint store (as in Constraint Logic Programming [9])
- $PSIC$ is a set of implications, representing *partially solved social integrity constraints*
- **PEXP** is the set of pending expectations
- **HAP** is the history of happened events
- **FULF** is a set of fulfilled expectations
- **VIOL** is a set of violated expectations

Initial Node and Success. A derivation D is a sequence of nodes $T_0 \rightarrow T_1 \rightarrow \ldots \rightarrow T_{n-1} \rightarrow T_n$. Given a goal G, a set of integrity constraints $\mathcal{IC}_S$, and an initial history $\mathbf{HAP}^i$ (that is typically empty) the first node is: $T_0 \equiv \langle \{G\}, \emptyset, \mathcal{IC}_S, \emptyset, \mathbf{HAP}^i, \emptyset, \emptyset \rangle$ i.e., the conjunction R is initially the query ($R_0 = \{G\}$) and the partially solved integrity constraints $PSIC$ is the whole set of social integrity constraints ($PSIC_0 = \mathcal{IC}_S$). The other nodes $T_j, j > 0$, are obtained by applying one of the transitions of the proof procedure, until no further transition can be applied (we call this last condition *quiescence*).

Let us now give the definition of successful derivation, both in the case of an open society instance (where new events may be added to the history further on) and of a closed society instance.

Definition 9. *Given an initial history* $\mathbf{HAP}^i$ *that evolves toward a final history* $\mathbf{HAP}^f$ *(with* $\mathbf{HAP}^f \supseteq \mathbf{HAP}^i$*), and an open society instance* $\mathcal{S}_{\mathbf{HAP}^i}$*, there exists an* open successful derivation *for a goal G iff the proof tree with root node*

$$\langle \{G\}, \emptyset, \mathcal{IC}_S, \emptyset, \mathbf{HAP}^i, \emptyset, \emptyset \rangle$$

has at least one leaf node

$$\langle \emptyset, CS, PSIC, \mathbf{PEXP}, \mathbf{HAP}^f, \mathbf{FULF}, \emptyset \rangle$$

where CS is consistent (i.e., there exists a ground variable assignment such that all the constraints are satisfied).

Analogously, there exists a closed successful derivation *iff the proof tree has at least one leaf node*

$$\langle \emptyset, CS, PSIC, \mathbf{PEXP}, \overline{\mathbf{HAP}^f}, \mathbf{FULF}, \emptyset \rangle$$

where CS is consistent, and $\mathbf{PEXP}$ *contains only negative literals* $\neg\mathbf{E}$ *and* $\neg\mathbf{EN}$.

From each non-failure leaf node N, answers can be extracted in a very similar way to the IFF proof procedure. Answers of the $\mathcal{S}$CIFF proof procedure, called *expectation answers*, are composed of an answer substitution and a set of abduced expectations. First, an answer substitution σ' is computed such that (*i*) σ' replaces all variables in N that are not universally quantified by a ground term (*ii*) σ' satisfies all the constraints in the store CS_N. Notice that, by definition 9, there must be a grounding of the variables satisfying all the constraints. In other words, we assume that the solver is (theory) complete [21], i.e., for each set of constraints c, the solver always returns *true* or *false*, and never *unknown*. Otherwise, if the solver is incomplete, σ' may not exist. The non-existence of σ' is discovered during the answer extraction phase. In such a case, the node N will be marked as a failure node, and another leaf node can be selected (if it exists).

Definition 10. *Given a non-failure node N, let σ' be the answer substitution extracted from N.*

Let $\sigma = \sigma'|_{vars(G)}$ be the restriction of σ' to the variables occurring in the initial goal G. Let $\mathbf{EXP}_N = (\mathbf{FULF}_N \cup \mathbf{PEXP}_N)\sigma'$. The pair $(\mathbf{EXP}_N, \sigma)$ is the expectation answer *obtained from the node N.*

3.1 Transitions

The transitions are based on those of the IFF proof procedure, augmented with those of CLP [9], and with specific transitions accommodating the concepts of fulfillment, dynamically growing history and consistency of the set of expectations with respect to the given definitions (Definitions 2, 3, and 4).

Due to lack of space, we do not list all the transitions, but we informally describe the main ones, and we give the formal definition of one (*Violation* **EN**), in order to give the taste of how the proof procedure works. The full list of transition can be found in [10].

IFF-Like Transitions. The $\mathcal{S}$CIFF proof procedure inherits the transitions of the IFF proof procedure. The IFF proof procedure starts with a formula (that replaces the concept of *resolvent* in logic programming) built as a conjunction of the initial query and the *ICs*. Then it repeatedly applies one of the following *inference rules*:

Unfolding replaces resolution: given a node with a definite atom, it replaces it with one of its definitions;

Propagation propagates ic_S: given a node containing $A \wedge B \rightarrow C$ and an atom A' that unifies with A, it replaces the implication with $(A = A') \wedge B \rightarrow C$;

Splitting distributes conjunctions and disjunctions, making the final formula in a sum-of-products form;

Case Analysis if the body of an ic_S contains $A = A'$, case analysis nondeterministically tries $A = A'$ or $A \neq A'$,

Factoring tries to reuse a previously made hypothesis;

Rewrite Rules for Equality use the inferences in the Clark Equality Theory;

Logical Simplifications try to simplify a formula through equivalences like $[A \wedge false] \leftrightarrow false$, $[true \rightarrow A] \leftrightarrow A$, etc.

Thanks to these inference rules, each node is always translated into a (disjunction of) conjunctions of atoms and implications; e.g., it can look like:

$$\begin{gathered}(A_1 \wedge A_2 \wedge [B_1 \wedge B_2 \rightarrow A_3] \wedge [B_3 \wedge B_4 \rightarrow A_4]) \\ \vee\ (A_i \wedge A_j \wedge A_k \wedge [B_y \rightarrow A_z] \wedge [B_5 \rightarrow false])\end{gathered}$$

the atoms have a similar meaning to those in the resolvent in LP, while the implications are (partially-propagated) integrity constraints.

Given a formula, it is always clear the quantification of the variables by the following rules:

- if a variable is in the initial query, then it is *free*;
- *else* if it appears in an atom, it is existentially quantified;
- *else* (it appears only in implications) it is universally quantified.

CLP-Like. The $\mathcal{S}$CIFF proof procedure also deals with constraints. It contains the CLP transitions [9] of *Constrain* (moves a constraint from R to the constraint store CS), *Infer* (infers new constraints given the current state of CS) and *Consistent* (checks if the constraint store is satisfiable). The solver has been extended to deal with unification and disunification of existentially and universally quantified atoms.

Dynamically Incoming Events. We assume to have an external set of events that happen in the society; the events in this external set are inserted in the history **HAP** by a transition *Happening.* Other transitions deal with non-happening of events and closure of the history.

Consistency, Fulfillment and Violation. In order to rule out nodes that are either inconsistent with respect to the declarative semantics or contain violations, we defined transitions that nondeterministically try to unify/disunify the terms in atoms. For instance, in order to detect a violation of **EN** atoms, we need to check if one happened event unifies with it. We have the transition:

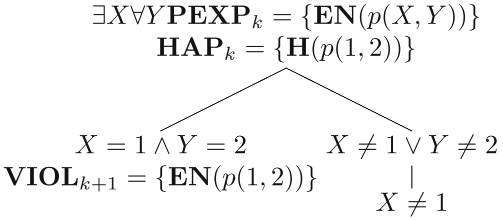
Violation **EN** Given a node N_k as follows:

- $\mathbf{PEXP}_k = \mathbf{PEXP'} \cup \{\mathbf{EN}(E_1)\}$
- $\mathbf{HAP}_k = \mathbf{HAP'} \cup \{\mathbf{H}(E_2)\}$

violation **EN** produces two nodes N^1_{k+1} and N^2_{k+1}, where

N^1_{k+1}			N^2_{k+1}		
$\mathbf{VIOL}_{k+1}$	$=$	$\mathbf{VIOL}_k \cup \{\mathbf{EN}(E_1)\}$	$\mathbf{VIOL}_{k+1}$	$=$	$\mathbf{VIOL}_k$
CS_{k+1}	$=$	$CS_k \cup \{E_1 = E_2\}$	CS_{k+1}	$=$	$CS_k \cup \{E_1 \neq E_2\}$

Example 1. Suppose that $\mathbf{HAP}_k = \{\mathbf{H}(p(1,2))\}$ and $\exists X \forall Y \mathbf{PEXP}_k = \{\mathbf{EN}(p(X,Y))\}$. *Violation* **EN** will produce the two following nodes:


$\exists X \forall Y \mathbf{PEXP}_k = \{\mathbf{EN}(p(X,Y))\}$
$\mathbf{HAP}_k = \{\mathbf{H}(p(1,2))\}$
$X = 1 \wedge Y = 2$
$\mathbf{VIOL}_{k+1} = \{\mathbf{EN}(p(1,2))\}$
$X \neq 1 \vee Y \neq 2$
$X \neq 1$

where the last simplification in the right branch is due to the rules of the constraint solver [10].

3.2 Sample Derivation

Let us consider a simple protocol: if a computee is asked for some information, it should either provide the information or refuse, but not both.[3] The protocol definition is given by means of the following Social Integrity Constraints:

$$
\begin{aligned}
IC_1\colon\ & \mathbf{H}(tell(A, B, query\text{-}ref(Info), D), T) \Rightarrow \\
& \mathbf{E}(tell(B, A, inform(Info, Answer), D), T_1), T_1 < T + 10 \vee \\
& \mathbf{E}(tell(B, A, refuse(Info), D), T_1), T_1 < T + 10 \\
IC_2\colon\ & \mathbf{H}(tell(A, B, inform(Info, Answer), D), T) \Rightarrow \\
& \mathbf{EN}(tell(A, B, refuse(Info), D), T_1), T_1 > T \\
IC_3\colon\ & \mathbf{H}(tell(A, B, refuse(Info), D), T) \Rightarrow \\
& \mathbf{EN}(tell(A, B, inform(Info, Answer), D), T_1), T_1 > T
\end{aligned}
$$

and let us suppose that the history evolves from an empty history to a final history $\mathbf{HAP}^f$ composed of only two events:

$$
\begin{aligned}
& \mathbf{H}(tell(yves, david, query\text{-}ref(train\_info), d_1), 1) \\
& \mathbf{H}(tell(david, yves, inform(train\_info, \text{“}departs(sv,rm,10{:}15)\text{”}), d_1), 2)
\end{aligned}
$$

The first node of the derivation tree is $N_0 \equiv \langle \emptyset, \emptyset, \mathcal{IC}_S, \emptyset, \emptyset, \emptyset, \emptyset \rangle$. The only applicable transition is *Happening* with one of the events in the external set of happened events; in this example, we will consider the events in chronological order:

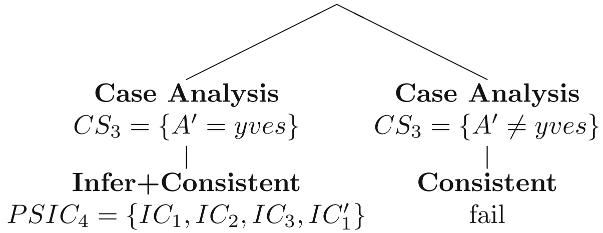
[3] This protocol is inspired to the FIPA query-ref interaction protocol [22].

$N_1 \equiv \langle \emptyset, \emptyset, PSIC, \emptyset, \{\mathbf{H}(\mathit{tell}(\mathit{yves}, \mathit{david}, \mathit{query\text{-}ref}(\mathit{train\_info}), d_1), 1)\}, \emptyset, \emptyset \rangle$.

Now transition *Propagation* is applicable to IC_1.

$$\begin{aligned} PSIC_2 = \{\ & IC_1, IC_2, IC3, \\ & A' = yves, B' = david, Info' = train\_info, D' = d_1, T' = 1 \\ & \rightarrow \mathbf{E}(\mathit{tell}(B', A', \mathit{inform}(\mathit{Info}', \mathit{Answer}'), D'), T_1'),\ T_1' < T' + 10 \\ & \vee\ \ \mathbf{E}(\mathit{tell}(B', A', \mathit{refuse}(\mathit{Info}'), D'), T_1'), \qquad T_1' < T' + 10 \\ & \} \end{aligned}$$

Each of the equalities in the body of the implication is dealt with by *case analysis*. Concerning $A' = yves$, case analysis generates two nodes: in the first $A' = yves$ and in the second $A' \neq yves$ is put in the constraint store. Since A' is universally quantified, the constraint $A' = yves$ succeeds when applying transition *Consistent*, and $A' \neq yves$ fails.



where

$$IC_1' = \begin{cases} B' = david, Info' = train\_info, D' = d_1, T' = 1 \\ \qquad \rightarrow \mathbf{E}(\mathit{tell}(B', \mathit{yves}, \mathit{inform}(\mathit{Info}', \mathit{Answer}'), D'), T_1'), T_1' < T' + 10 \\ \qquad \vee\, \mathbf{E}(\mathit{tell}(B', \mathit{yves}, \mathit{refuse}(\mathit{Info}'), D'), T_1'), T_1' < T' + 10 \end{cases}$$

After applying case analysis for each equality in the body, and the successive constraint solving step, we have only one non-failure node:

$$N_{10} = \langle \emptyset, \emptyset, PSIC_{10}, \emptyset, \mathbf{HAP}_{10}, \emptyset, \emptyset \rangle$$

$$\begin{aligned} PSIC_{10} &= \{IC_1, IC_2, IC_3, \\ &\qquad \mathit{true} \rightarrow \mathbf{E}(\mathit{tell}(\mathit{david}, \mathit{yves}, \mathit{inform}(\mathit{train\_info}, \mathit{Answer}'), d_1), T_1'), \\ &\qquad\qquad T_1' < 1 + 10 \\ &\qquad\qquad \vee\ \ \mathbf{E}(\mathit{tell}(\mathit{david}, \mathit{yves}, \mathit{refuse}(\mathit{train\_info}), d_1), T_1'), T_1' < 1 + 10\} \\ \mathbf{HAP}_{10} &= \{\mathbf{H}(\mathit{tell}(\mathit{yves}, \mathit{david}, \mathit{query\text{-}ref}(\mathit{train\_info}), d_1), 1)\} \end{aligned}$$

We apply *Logical Equivalence* to the implication with *true* antecedent. Then, since element R of the produced node (N_{11} not shown here) contains a disjunction, *splitting* can be applied, and its application generates two nodes. Let us consider the first node N_{14}', having:

$$\begin{aligned} R_{14}' &= \emptyset \\ \mathbf{PEXP}_{14}' &= \{\mathbf{E}(\mathit{tell}(\mathit{david}, \mathit{yves}, \mathit{inform}(\mathit{train\_info}, \mathit{Answer}'), d_1), T_1')\} \\ CS_{14}' &= \{T_1' < 1 + 10\} \end{aligned}$$

The declarative reading of this node is:

$\exists Answer', \exists T_1'.\ \ T_1' < 1 + 10$
$\wedge\ \mathbf{E}(\mathit{tell}(\mathit{david},\ \mathit{yves},\ \mathit{inform}(\mathit{train\_info}, \mathit{Answer}'), d_1), T_1')$.

Suppose that now *happening* transition is applied with the second event in the external set of happened events[4].

$$\begin{aligned}\mathbf{HAP}_{15} = \{&\mathbf{H}(\mathit{tell}(\mathit{yves}, \mathit{david}, \mathit{query\text{-}ref}(\mathit{train\_info}), d_1), 1),\\ &\mathbf{H}(\mathit{tell}(\mathit{david}, \mathit{yves}, \mathit{inform}(\mathit{train\_info}, \text{“\textit{departs(sv,rm,10:15)}”}, d_1), 2).\}\end{aligned}$$

We can now apply transition *fulfillment* **E** with the event $\mathbf{H}(\mathit{tell}(\mathit{david}, \mathit{yves}, \mathit{inform}\ldots))$ in the history. The transition opens two alternative nodes, N'_{16} and N''_{16}: either the event in the expectation unifies with the event in the history, and becomes fulfilled, or it does not unify and remains pending.

$$\begin{aligned}
CS'_{16} &= \{\mathit{Answer}' = \text{“\textit{departs(sv,rm,10:15)}”} \wedge T'_1 = 2\\
&\quad \wedge T'_1 < 1 + 10\}\\
\mathbf{FULF}'_{16} &= \{\mathbf{E}(\mathit{tell}(\mathit{david}, \mathit{yves}, \mathit{inform}(\mathit{train\_info}, \mathit{Answer}'), d_1), T'_1)\}\\
\mathbf{PEXP}'_{16} &= \emptyset\\
CS''_{16} &= \{(\mathit{Answer}' \neq \text{“\textit{departs(sv,rm,10:15)}”} \vee T'_1 \neq 2)\\
&\quad \wedge T'_1 < 1 + 10\}\\
\mathbf{FULF}''_{16} &= \emptyset\\
\mathbf{PEXP}''_{16} &= \{\mathbf{E}(\mathit{tell}(\mathit{david}, \mathit{yves}, \mathit{inform}(\mathit{train\_info}, \mathit{Answer}'), d_1), T'_1)\}
\end{aligned}$$

The second node can be fulfilled if the history is still open, as other events may happen matching the pending expectation. If the history gets closed, the pending expectation will become violated, so the second will be a violation node. This does not mean that the proof is in a global violation. As in SLD resolution a global failure is obtained only if all the leaves of the proof tree are failure nodes, in the same way in $\mathcal{S}$CIFF we have a global violation only if all the leaves contain violations (i.e., in all alternative branches, $\mathbf{VIOL} \neq \emptyset$). This is not the case in this example, since in the first node, N'_{16}, the expectations are fulfilled).

Other transitions are applicable to this node; we do not continue the example because their application is very similar to the ones already presented. For example, transition *Propagation* will be applied to IC_2 and to the event $\mathbf{H}(\mathit{tell}(\mathit{david}, \mathit{yves}, \mathit{inform}\ldots))$ in the history, thus providing a new expectation $\mathbf{EN}(\mathit{tell}(\mathit{david}, \mathit{yves}, \mathit{refuse}(\mathit{train\_info}), d_1), T'_1)$, $T'_1 > 2$.

4 Implementation

In this section, we describe the implementation of SOCS-SI, the tool for compliance verification of agent interaction. The tool is composed of an implementation of the $\mathcal{S}$CIFF proof-procedure specified in the previous section, interfaced to a graphical user interface and to a component for the observation of agent interaction.

The SOCS-SI software application is composed of a set of modules. All the components except one ($\mathcal{S}$CIFF) are implemented in the Java language.

[4] Of course, the *happening* transition was applicable also to the previous nodes. We are giving here a sample derivation, but others may be possible.

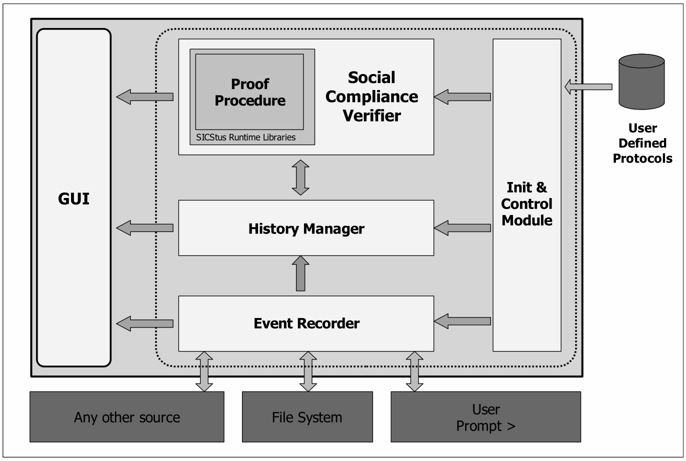


Fig. 1. Overview of the *SOCS-SI* architecture

The core of SOCS-SI is composed of three main modules (see Fig. 1), namely:

- *Event Recorder*: fetches events from different sources and stores them inside the *History Manager*.
- *History Manager*: receives events from the *Event Recorder* and composes them into an "event history".
- *Social Compliance Verifier*: fetches events from the *History Manager* and passes them to the proof-procedure in order to check the compliance of the history to the specification.

Computees communicate by exchanging messages, which are then translated into **H** events. The *Event Recorder* fetches events and records them into the *History Manager*, where they become available to the proof-procedure (see Sect. 4.1). As soon as the proof-procedure is ready to process a new event, it fetches one from the *History Manager*. The event is processed and the results of the computation are returned to the GUI. The proof-procedure then continues its computation by fetching another event if there is any available, otherwise it suspends, waiting for new events.

A fourth module, named *Init&Control Module* provides for initialization of all the components in the proper order. It receives as initial input a set of protocols defined by the user, which will be used by the proof-procedure in order to check the compliance of agents to the specification.

4.1 Implementation of $\mathcal{S}$CIFF

The $\mathcal{S}$CIFF proof procedure has been implemented in SICStus Prolog [23], exploiting its constraint libraries and, in particular, the *Constraint Handling Rules* (CHR) library [24].

The data structures representing the proof tree nodes are represented as follows:

- *R* is represented by the Prolog resolvent;
- *CS* is the CLP (CLPFD, CLPB) constraint store;
- *PSIC*, **EXP**, **HAP**, **FULF**, **VIOL** are represented as *CHR* constraints.

Attributes [25] are used to represent the quantification (existential or universal) of variables in expectations; an *ad-hoc CHR* constraint (`reif_unify/3`) implements reified unification (i.e., both the constraints = and ≠) between variables and terms.

Thanks to the representation of most data structures as *CHR* constraints, the transitions (such as propagation, happening, fulfillment/violation) that modify those data structures have been implemented by exploiting the *CHR* computational model.

For instance, the following rule immplements the check for **E**-consistence:

```
e_consistency @
        e(EEvent,ETime),
        en(ENEvent,ENTime)
        ==>
        reif_unify(p(EEvent,ETime),p(ENEvent,ENTime),0).
```

This is a *propagation* rule, i.e., a rule that adds a constraint to the *CHR* store whenever a combination of constraints is present in the store. The name of the rule is `e_consistency`. `e(EEvent,ETime)` and `en(ENEvent,ENTime)` are the two *CHR* constraints representing the expectations **E**(*EEvent*, *ETime*) and **EN**(*ENEvent*, *ENTime*), respectively. Whenever these two constraints are in the *CHR* store, the dis-unification constraint

```
reif_unify(p(EEvent,ETime),p(ENEvent,ENTime),0)
```

is added to the store to impose that the arguments of the positive and the negative expectations do not unify, as required by **E**-consistency (see Def. 3).

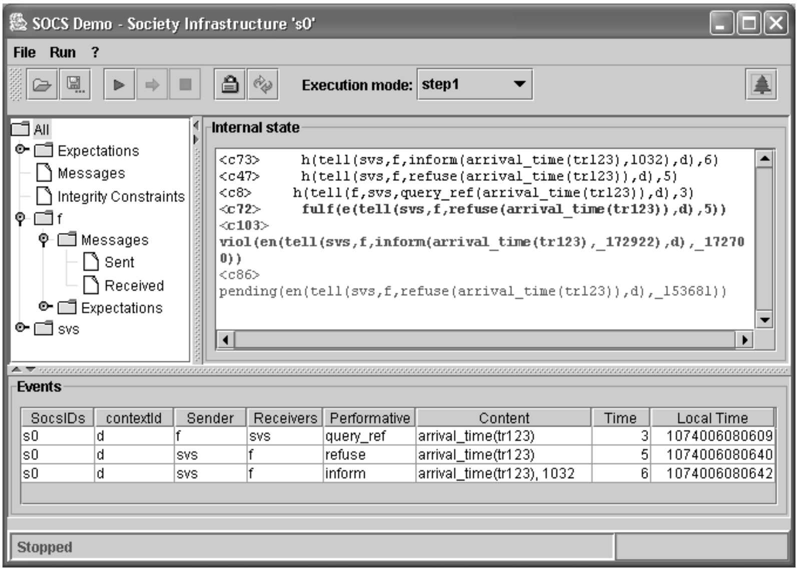
The CLP transitions, instead, are delegated to the CLP solvers available in SICStus Prolog: we have used CLPFD for finite domains and CLPB for binary domains variables, but in principle it would be possible to use any CLP library based on SICStus.

The $\mathcal{S}$CIFF proof tree is searched with a depth-first strategy, so to exploit the Prolog stack for backtracking. The success of the proof procedure (see Def. 8) is mapped onto a successful Prolog derivation.

4.2 The Graphical User Interface

The Graphical User Interface is implemented by using the Swing graphic library, and implements the Model-View-Control programming pattern. The main window is composed of three areas (or sub-windows), and of a button bar that contains the controls (see Fig. 2).

The bottom area contains the list of all the messages received by SOCS-SI. The left pane contains the list of agents known by the society, i.e., agents that have performed at least one communicative action. The central pane contains the



Execution mode: step1

Internal state

```
<c73>     h(tell(svs,f,inform(arrival_time(trl23),1032),d),6)
<c47>     h(tell(svs,f,refuse(arrival_time(trl23)),d),5)
<c8>     h(tell(f,svs,query_ref(arrival_time(trl23)),d),3)
<c72>     fulf(e(tell(svs,f,refuse(arrival_time(tr123)),d),5))
<c103>
viol(en(tell(svs,f,inform(arrival_time(tr123),_172922),d),_17270
0))
<c86>
pending(en(tell(svs,f,refuse(arrival_time(trl23)),d),_153681))
```

Events

SocsIDs	contextId	Sender	Receivers	Performative	Content	Time	Local Time
s0	d	f	svs	query_ref	arrival_time(tr123)	3	1074006080609
s0	d	svs	f	refuse	arrival_time(tr123)	5	1074006080640
s0	d	svs	f	inform	arrival_time(tr123), 1032	6	1074006080642

Fig. 2. A screenshot of the application

results of the computation, returned by the proof-procedure. These results are expressed in terms of society expectations about the future behavior of agents, and also in terms of fulfilled expectations and violations of social rules. By selecting an agent from the left pane, it is possible to restrict the information shown on the larger pane to be only that relevant to that particular agent. Among other features, it is possible to execute step-by-step the application, so that it elaborates one message at a time and then waits for a user acknowledge (similarly to the debug interface of modern compilers).

5 Related Work

The main result of the first two years of the SOCS project, with respect to *societies* od computees, is the definition of a social framework. In doing this, we have provided (*i*) a declarative representation of the social knowledge, (*ii*) a *logic formalism* based on social expectations and $\mathcal{IC}_S$, for *specifying* social rules and easily verifiable protocols, and for defining the semantics of communicative acts in an open system scenario, (*iii*) a *proof-procedure* proven correct with respect to the declarative representation of the social knowledge, and (*iv*) a prototypical *implementation* of the social framework which can be used to test the framework with a number of scenarios, protocols, and communication languages.

Our work relates to several aspects of Multi-Agent Systems research, in terms of social and interaction models, operational frameworks, and implemented systems, and to work done in Computational Logics, specifically in extensions of logic programming. Space limitations prevent us from thoroughly discussing here the SOCS social framework in relationship with other conspicuous work done on all these areas. We will only give an overview of some related work, and give a reference to the relevant project deliverables for further discussions.

$\mathcal{IC}_S$ represent in a way social norms. Several researchers have studied the concepts of norms, commitments and social relations in the context of Multi-Agent Systems [26]. Furthermore, a lot of research has been devoted in proposing architectures for developing agents with social awareness (see, for instance [27]). Our approach can be conceived as complementary to these efforts, since instead of proposing a specific architecture for designing computees, our work is mainly focused on the definition of a society infrastructure based on Computational Logic for regulating and improving robustness of interaction in an open environment, where the internal architecture of the computees might be unknown.

Our work is very related, as far as objectives and methodology, to the work on computational societies presented and developed in the context of the ALFEBIITE project [28]. We have in fact the same understanding of openness, as we pointed out in the introduction. In turn, our work is especially oriented to computational aspects, and it was developed with the purpose of providing a computational framework that can be directly used for automatic verification of properties such as compliance.

Most formal approaches to protocol modelling specify protocols as legal sequences of actions [29, pp.19–22]. In this way, protocols can be over-constrained, and this affects autonomy, heterogeneity, and ability to exploit opportunities and exceptions [30]. Moreover, the mentalistic approach to protocol definition has been much criticized mainly because its assumptions regarding agents' internals are not realistic in open societies of heterogeneous agents [31]. Therefore we advocated a *social* approach, where the semantics of interactions is defined in terms of the effects of the computees interactions on the society. Following this approach, even if the computees mental state cannot be accessed, it is possible to verify whether communicating computees in a society comply to some social laws and norms which regulate the interactions. Another expressive advantage of our framework is that it can express with the same formalism both protocols and social semantics of communicative acts.

In our social approach we drew inspiration from work done by Yolum and Singh [30] where a social semantics of agent interaction protocols is exemplified by using a commitment-based approach, and by Fornara and Colombetti [32–34], especially as it concerns the semantics of communication languages [29, pp. 37–41]. In particular, the latter approach is similar to ours in that it specifies the semantics of actions in terms of their social effects, and presupposes a social framework (which is called *institution* in [33]) for assigning agents with roles, verifying their social behavior and, possibly, recovering from violation conditions. There are, however, some significant differences with [33], mainly originating from the different paradigm we have chosen to express semantics (*logic-based* instead of *object-oriented*), as it is shown in [29, p. 71].

Yolum and Singh also propose an interesting way of linking together communicative acts and protocol specifications using the idea of social semantics in

[30], where an agent can find a communication path leaving no pending commitments by exploiting its reasoning/planning capabilities. Our approach rather aims at ensuring protocol compliance regardless of computees' reasoning capabilities. In fact, $\mathcal{IC}_S$ are designed to explicit constraints between communicative acts. However, equipping the communication model of single computees with sufficient knowledge to reason about social expectations is certainly an interesting option. This topic is discussed in D4 [35].

Finally, there exist other approaches based on Deontic Logic to formally defining norms and dealing with their possible violations. An example of it is work by Dignum et al. [36], as discussed in [29, pp. 72]. Deontic operators have been used not only at the social level, but also at the agent level. Notably, in IMPACT [37, 38], agent programs may be used to specify what an agent is obliged to do, what an agent may do, and what an agent cannot do on the basis of deontic operators of Permission, Obligation and Prohibition (whose semantics does not rely on a Deontic Logic semantics). In this respect, IMPACT and our work have similarities even if their purpose and expressivity are different. The main difference is that the goal of agent programs in IMPACT is to express and determine by its application the behavior of a single agent, whereas our goal is to express rules of interaction, that instead cannot really determine and constrain the behavior of the single computees participating to the interaction protocols, since computees are autonomous.

Our work is not only directly related to social aspects of MAS, but also to extensions of Logic Programming for MAS. In particular, the syntax of ic_S is strictly related to that of integrity constraints in the IFF proof-procedure [8]. In [29, pp. 92–94] work on $\mathcal{IC}_S$ is discussed with that done by Fung and Kowalski, with a focus on some syntactic aspects of the integrity constraints handled by the IFF proof-procedure. Briefly, the $\mathcal{S}$CIFF can be considered as an extension of the IFF proof procedure that also:

- abduces atoms with variables universally quantified;
- deals with CLP constraints, also imposed as quantifier restrictions on universally quantified variables;
- is more dynamic, in fact new events may arrive, and the proof procedure dynamically takes them into consideration in the knowledge base;
- has the new concepts, related to on-line verification, of *fulfillment* and *violation.*

The IFF is not the only abductive logic programming proof-procedure in literature. Various abductive proof procedures have been proposed in the past. In [39, p. 173] other procedures are discussed in relationship with our choice based on the IFF.

Other authors also proposed using abduction for verification. Noteworthily, Russo et al. [40] use an abductive proof procedure for analyzing event-based requirements specifications. In their approach, the system has a declarative specification given through the Event Calculus [41] axioms, and the goal is proving that some invariant I is true in all cases. This method uses abduction for analyzing the correctness of specifications, while our system is more focussed on the on-line check of compliance of a set of agents.

We will conclude this section by quickly mentioning two other implementations of social frameworks. The cited above ALFEBIITE project delivers a tool (*Society Visualiser*) to demonstrate animations of protocol runs in such systems [5]. The Society Visualiser's main purpose is to explicitly represent the institutional power of the members and the concept of valid action. As we stressed earlier on, our work is not based on any deontic infrastructure. For this reason, the SOCS social framework could be used in a different, possibly broader spectrum of application domains, ranging from intelligent agents to reactive systems.

ISLANDER [42] is a tool for the specification and verification of interaction in complex social infrastructures, such as electronic institutions. ISLANDER allows to analyse situations, called scenes, and visualise liveness or safeness properties in some specific settings. The kind of verification involved is static and is used to help designing institutions. Although our framework could also be used at design time, its main intended use is for on-the-fly verification of heterogeneous and open systems.

6 Conclusion and Future Work

In this work, we reported on a Computational Logic-based framework for modelling societies of computees and their interactions. We presented both published and original work done in the first two years of the SOCS project about modelling interactions among agents/computees. In [29, pp. 9–10] we give a list of pointers to publications where some of the results presented here can be seen in more detail.

One of the main objectives of SOCS was to deliver a formal logic-based framework to characterize the interactions between computees in a rule-based manner, either by relying on protocols shared and agreed upon by all computees in a given society, or by interaction patterns that are specific to individual computees and possibly different for different computees.

We defined a model which lent itself easily to a computational realisation, and which is precise and amenable to formal verification of properties pertaining to the interactions of the computees belonging to a particular societies.

The social model of interaction among computees that we propose follows a Computational Logic-based approach. In this model, Logic Programming, suitably extended with the concept of $\mathcal{IC}_S$ and expectations (interpreted as abducibles), acts as a uniform language for protocols, interaction policies and patterns.

A degree of openness, understood as the freedom of its members to join or leave the society, is given by the model presented in Section 2, which caters for new members joining a society and existing members leaving it [29, p. 27]. Another degree of openness, understood as the possibility to have a society of heterogeneous computees, is achieved by the fact that the model of the society, including the handling of expectations, the protocol conformance checking and the generation of violations, are only based on the socially observable behaviour of computees: no assumption is made on the internals of computees, but their social behaviour is constrained by the semantics of social actions and protocols. Non-conforming behaviour of computees is still possible, but it will be detected by the society infrastructure and it will have social consequences.

Violation handling and recovery is a matter of current and future work. The SOCS model of society caters for reasoning under incomplete information, in the sense that events that did not happen or that have not been "detected" are treated as unattended expectations, and it is possible to reason over both expectations and happened events.

The formalism for expressing society rules and protocols, together with the semantics of the individual communication utterances, is based on Abductive Logic Programming and constraints over abducible predicates, and its declarative semantics has been given in terms of logical entailment. The $\mathcal{S}$CIFF provides the operational support for the underlying infrastructure.

Time is explicit in the model. The "reasoning" at a social level is made over time, and it takes into account issues such as dealing with deadlines, that are important also from a practical viewpoint [29, pp. 35–36]. In this way, we propose a social framework which is suitable for modelling a dynamic setting and able to handle changes in a dynamic environment.

In [43] we present an evaluation of the society model in the context of the Global Computing programme. We believe that one of the strong points of our approach are to be found in its formality, not only at a syntactic level (definition of what is the format of the society knowledge, ic_S, protocols, CCL format and constraints), but also, and more interestingly, at the semantic level, which allows us to describe what are the desirable evolutions of a society and link these formally to the social structure and social behaviour of the computees.

Most importantly, the formality of the framework is indeed backed-up by an existing and well-defined operational counterpart, the $\mathcal{S}$CIFF, a proof-procedure which has been proven correct with respect to the model, implemented, and integrated in the implementation of a tool, SOCS-SI, which can be used to run some tests.

We have interpreted the protocol conformance checks and the normative control performed by the society as abductive tasks, and defined an extension of the IFF abductive proof procedure to deal with this task. The extension is non-trivial, and deals with complex forms of variables quantification in abductive logic programs, as well as constraint predicates.

Finally, we believe that a further contribution of the work presented in this document is that the computational models devised within SOCS for the computee and for their societies are can be easily integrated with each other, since they are based on a similar formalism and on the same technology.

Future work will go in the direction of testing the system in different scenarios and studying properties of the model and of specific instances of societies. Among the scenarios that we are considering to evaluate the expressiveness of the designed interaction models are: dialogue-based interaction, with a special focus on resource reallocation, a combinatorial auction and an electronic payment network protocol.

Acknowledgements

This work is partially funded by the Information Society Technologies programme of the European Commission under the IST-2001-32530 SOCS Project [3], and by the MIUR COFIN 2003 projects *Sviluppo e verifica di sistemi multiagente basati sulla logica* [44], and *La Gestione e la negoziazione automatica dei diritti sulle opere dell'ingegno digitali: aspetti giuridici e informatici.*

References

1. Bracciali, A., Demetriou, N., Endriss, U., Kakas, A., Lu, W., Mancarella, P., Sadri, F., Stathis, K., Terreni, G., Toni, F.: The KGP model of agency: Computational model and prototype implementation. In this volume.
2. Global Computing, Future and Emerging Technologies: Co-operation of Autonomous and Mobile Entities in Dynamic Environments. Home Page: `http://www.cordis.lu/ist/fetgc.htm`.
3. Societies Of ComputeeS (SOCS): a computational logic model for the description, analysis and verification of global and open societies of heterogeneous computees. Home Page: `http://lia.deis.unibo.it/research/socs/`.
4. Hewitt, C.: Open information systems semantics for distributed artificial intelligence. Artificial Intelligence **47** (1991) 79–106
5. Artikis, A., Pitt, J., Sergot, M.: Animated specifications of computational societies. In Castelfranchi, C., Lewis Johnson, W., eds.: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part III, Bologna, Italy, ACM Press (2002) 1053–1061
6. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Modeling interactions using *Social Integrity Constraints*: A resource sharing case study. In Leite, J.A., Omicini, A., Sterling, L., Torroni, P., eds.: Declarative Agent Languages and Technologies. Lecture Notes in Artificial Intelligence **2990**. Springer-Verlag (2004) 243–262
7. Kakas, A.C., Kowalski, R.A., Toni, F.: The role of abduction in logic programming. In Gabbay, D.M., Hogger, C.J., Robinson, J.A., eds.: Handbook of Logic in Artificial Intelligence and Logic Programming. Volume 5., Oxford University Press (1998) 235–324
8. Fung, T.H., Kowalski, R.A.: The IFF proof procedure for abductive logic programming. Journal of Logic Programming **33** (1997) 151–165
9. Jaffar, J., Maher, M.: Constraint logic programming: a survey. Journal of Logic Programming **19-20** (1994) 503–582
10. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Specification and verification of interaction protocols: a computational logic approach based on abduction. Technical Report CS-2003-03, Dipartimento di Ingegneria di Ferrara, Ferrara, Italy (2003) Available at `http://www.ing.unife.it/aree_ricerca/informazione/cs/technical_reports`.
11. Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: Compliance verification of agent interaction: a logic-based tool. In Trappl, R., ed.: Proceedings of the 17th European Meeting on Cybernetics and Systems Research, Vol. II, Symposium "From Agent Theory to Agent Implementation" (AT2AI-4), Vienna, Austria, Austrian Society for Cybernetic Studies (2004) 570–575

12. Stathis, K., Kakas, A.C., Lu, W., Demetriou, N., Endriss, U., Bracciali, A.: PROSOCS: a platform for programming software agents in computational logic. In Trappl, R., ed.: Proceedings of the 17th European Meeting on Cybernetics and Systems Research, Vol. II, Symposium "From Agent Theory to Agent Implementation" (AT2AI-4), Vienna, Austria, Austrian Society for Cybernetic Studies (2004) 523–528
13. Lloyd, J.W.: Foundations of Logic Programming. 2nd extended edn. Springer-Verlag (1987)
14. Torroni, P., Alberti, M., Chesani, F., Gavanelli, M., Lamma, E., Mello, P.: A logic based approach to interaction design in open multi-agent systems. In: Proceedings of the 13th IEEE international Workshops on Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE-2004), 2nd international workshop "Theory and Practice of Open Computational Systems (TAPOCS)", Modena, Italy, IEEE Press (2004) to appear.
15. Alberti, M., Gavanelli, M., Lamma, E., Mello, P., Torroni, P.: An Abductive Interpretation for Open Societies. In Cappelli, A., Turini, F., eds.: AI*IA 2003: Advances in Artificial Intelligence, Proceedings of the 8th Congress of the Italian Association for Artificial Intelligence, Pisa. Lecture Notes in Artificial Intelligence **2829**. Springer-Verlag (2003) 287–299
16. Cox, P.T., Pietrzykowski, T.: Causes for events: Their computation and applications. In: Proceedings CADE-86. (1986) 608–621
17. Eshghi, K., Kowalski, R.A.: Abduction compared with negation by failure. In Levi, G., Martelli, M., eds.: Proceedings of the 6th International Conference on Logic Programming, MIT Press (1989) 234–255
18. Kakas, A.C., Mancarella, P.: On the relation between Truth Maintenance and Abduction. In Fukumura, T., ed.: Proceedings of the 1st Pacific Rim International Conference on Artificial Intelligence, PRICAI-90, Nagoya, Japan, Ohmsha Ltd. (1990) 438–443
19. Poole, D.L.: A logical framework for default reasoning. Artificial Intelligence **36** (1988) 27–47
20. Kunen, K.: Negation in logic programming. In: Journal of Logic Programming. Volume 4. (1987) 289–308
21. Jaffar, J., Maher, M., Marriott, K., Stuckey, P.: The semantics of constraint logic programs. Journal of Logic Programming **37(1-3)** (1998) 1–46
22. FIPA Query Interaction Protocol (2001) Published on August 10th, 2001. Available for download from the FIPA web site, `http://www.fipa.org`.
23. SICStus prolog user manual, release 3.11.0 (2003) Available for download from the SICS web site, `http://www.sics.se/isl/sicstus/`.
24. Frühwirth, T.: Theory and practice of constraint handling rules. Journal of Logic Programming **37** (1998) 95–138
25. Holzbaur, C.: Specification of constraint based inference mechanism through extended unification. Dissertation, Dept. of Medical Cybernetics & AI, University of Vienna (1990)
26. Conte, R., Falcone, R., Sartor, G.: Special issue on agents and norms. Artificial Intelligence and Law **1** (1999)
27. Castelfranchi, C., Dignum, F., Jonker, C., Treur, J.: Deliberative normative agents: Principles and architecture. In Jennings, N.R., Lespérance, Y., eds.: Intelligent Agents VI. Lecture Notes in Computer Science **1757**. Springer-Verlag (1999) 364–378

28. ALFEBIITE: A Logical Framework for Ethical Behaviour between Infohabitants in the Information Trading Economy of the universal information ecosystem. IST-1999-10298 (1999) Home Page: `http://www.iis.ee.ic.ac.uk/~alfebiite/ab-home.htm`.
29. Mello, P., Torroni, P., Gavanelli, M., Alberti, M., Ciampolini, A., Milano, M., Roli, A., Lamma, E., Riguzzi, F., Maudet, N.: A logic-based approach to model interaction amongst computees. Technical report, SOCS Consortium (2003) Deliverable D5. Available on request.
30. Yolum, P., Singh, M.: Flexible protocol specification and execution: applying event calculus planning using commitments. In Castelfranchi, C., Lewis Johnson, W., eds.: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part II, Bologna, Italy, ACM Press (2002) 527–534
31. Singh, M.: Agent communication language: rethinking the principles. IEEE Computer (1998) 40–47
32. Fornara, N., Colombetti, M.: Operational specification of a commitment-based agent communication language. In Castelfranchi, C., Lewis Johnson, W., eds.: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part II, Bologna, Italy, ACM Press (2002) 535–542
33. Colombetti, M., Fornara, N., Verdicchio, M.: The role of institutions in multiagent systems. In: Proceedings of the Workshop on Knowledge based and reasoning agents, VIII Convegno AI*IA 2002, Siena, Italy. (2002)
34. Colombetti, M., Fornara, N., Verdicchio, M.: A social approach to communication in multiagent systems. In Leite, J.A., Omicini, A., Sterling, L., Torroni, P., eds.: Declarative Agent Languages and Technologies. Lecture Notes in Artificial Intelligence **2990**. Springer-Verlag (2004) 193–222
35. Kakas, A., Mancarella, P., Sadri, F., Stathis, K., Toni, F.: A logic-based approach to model computees. Technical report, SOCS Consortium (2003) Deliverable D4. Available on request.
36. Dignum, V., Meyer, J.J., Dignum, F., Weigand, H.: Formal specification of interaction in agent societies. In: Proceedings of the Second Goddard Workshop on Formal Approaches to Agent-Based Systems (FAABS), Maryland. (2002)
37. Arisha, K.A., Ozcan, F., Ross, R., Subrahmanian, V.S., Eiter, T., Kraus, S.: IMPACT: a Platform for Collaborating Agents. IEEE Intelligent Systems **14** (1999) 64–72
38. Eiter, T., Subrahmanian, V., Pick, G.: Heterogeneous active agents, I: Semantics. Artificial Intelligence **108** (1999) 179–255
39. Kakas, A.C., Lamma, E., Mancarella, P., Mello, P., Stathis, K., Toni, F.: Computational model for computees and societies of computees. Technical report, SOCS Consortium (2003) Deliverable D8. Available on request.
40. Russo, A., Miller, R., Nuseibeh, B., Kramer, J.: An abductive approach for analysing event-based requirements specifications. In Stuckey, P., ed.: Logic Programming, 18th International Conference, ICLP 2002. Lecture Notes in Computer Science **2401**. Springer-Verlag (2002) 22–37
41. Kowalski, R.A., Sergot, M.: A logic-based calculus of events. New Generation Computing **4** (1986) 67–95
42. Esteva, M., de la Cruz, D., Sierra, C.: ISLANDER: an electronic institutions editor. In Castelfranchi, C., Lewis Johnson, W., eds.: Proceedings of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2002), Part III, Bologna, Italy, ACM Press (2002) 1045–1052

43. Bracciali, A., Kakas, A.C., Lamma, E., Mello, P., Stathis, K., Toni, F., Torroni, P.: D11: Evaluation and self assessment. Technical report, SOCS Consortium (2003) Deliverable D11. Available on request.
44. MASSiVE: sviluppo e verifica di sistemi multi-agente basati sulla logica. Home Page: `http://www.di.unito.it/massive/`.

The KGP Model of Agency for Global Computing: Computational Model and Prototype Implementation

A. Bracciali[1], N. Demetriou[2], U. Endriss[3], A. Kakas[2], W. Lu[4], P. Mancarella[1],
F. Sadri[3], K. Stathis[4,1], G. Terreni[1], and F. Toni[3,1]

[1] Dip. di Informatica, Università di Pisa
{braccia, paolo, terreni}@di.unipi.it
[2] Dept of Computer Science, Cyprus University
{demetriou, antonis}@cs.ucy.ac.cy
[3] Dept of Computing, Imperial College London
{ue, fs, ft}@doc.ic.ac.uk
[4] School of Informatics, City University London
{lue, kostas}@soi.city.ac.uk

Abstract. We present the computational counterpart of the KGP (**K**nowledge, **G**oals, **P**lan) declarative model of agency for Global Computing. In this context, a computational entity is seen as an agent developed using Computational Logic tools and techniques. We model a KGP agent by relying upon a collection of capabilities, which are then used to define a collection of transitions, to be used within logically specified, context sensitive control theories, which we call cycle theories. In close relationship to the declarative model, the computational model mirrors the logical architecture by specifying appropriate computational counterparts for the capabilities and using these to give the computational models of the transitions. These computational models and the one specified for the cycle theories are all based on, and are significant extensions of, existing proof procedures for abductive logic programming and logic programming with priorities. We also discuss a prototype implementation of the overall computational model for KGP.

1 Introduction

Global Computing (GC) and its applications rely upon computing environments that are composed of *autonomous* computational entities whose activity is not centrally controlled but is *decentralised* instead. Decentralisation results either because global control is impossible or at times impractical, or because the entities are created or controlled by different owners. The computational entities may be mobile, due to the movement of the physical platforms or by movement of the entities from one platform to another. In other words, the environment in which the entities are situated is *open* and evolves over time. For instance, in a typical GC application it might be required to allow for the introduction

C. Priami and P. Quaglia (Eds.): GC 2004, LNCS 3267, pp. 340–367, 2004.

and deletion of computational entities. The internal structure and behaviour of these entities may also be *heterogeneous* and may vary over time.

Programming the behaviour of a computational entity that is situated in a GC environment is a non-trivial task. One of the problems is that such an entity should be in a position to operate with *incomplete information* about the environment. Incompleteness might arise from the entities having newly joined the environment of an application and having only a partial view over the status of that application. Incompleteness might also arise from the autonomy of the entities and their unwillingness to disclose information about themselves. Moreover, incompleteness might sometimes be caused by the fact that information in a GC environment becomes rapidly out of date. Thus, a GC entitity needs to be able to discover relevant information or other entities in the dynamically evolving environment.

If the ultimate goal of GC research is to provide a solid scientific foundation for the design of GC systems, we will need to lay the groundwork for achieving effective principles for building and analysing such systems. In trying to achieve this goal, within the GC project SOCS we interpret the GC vision as follows. Entities in GC systems are defined via *Computational Logic* (CL), as understood in [26, 29, 27], which is used to define their internal organisation, reasoning and their mutual interactions. We call the entities *computees*, standing for *agents in CL*[1]. One important feature of computees is that they are able to *reason* by using CL tools and techniques. We call the systems composed of such entities *societies* (see [7]) as they are characterised by "social rules" for computees to interact and operate in the presence of each other.

In order to interact freely, computees can use *high-level communication*, as understood in multi-agent systems. Computees may be *heterogeneous* as far as *behaviour* is concerned, provided by CL-based cycle theories allowing a highly modular and flexible specification of control. Cycle theories allow to render computees adaptable to dynamically changing environments and allow to characterise, via different cycle theories, heterogeneously behaved computees.

Computees also need to adapt their internal state as the environments in which they are situated evolve. A number of CL techniques have been developed for addressing tasks such as temporal reasoning in a changing environment, hypothetical reasoning for dealing with incomplete information, hypothetical reasoning for planning, hypothetical reasoning to achieve communication, argumentation for decision-making, inductive logic programming for learning. However, in order to cope with the GC challenges, CL techniques in isolation are inadequate, as none serves all dimensions in the operation of computees. Our model for computees integrates (extensions of) a number of existing CL techniques, in order to achieve the enhanced performance which is required by the GC vision.

We call our model *KGP*, since computees' internal state consists of a knowledge base (K), from which they reason, goals (G) that they need to achieve,

[1] In this paper, we will use the terms computees and agents interchangeably.

and plans (P) for their goals, consisting of actions that may be physical, sensing or communicative. Computees pursue their goals while being alert to the environment and adapt their goals and plan to any changes that they perceive.

The paper is organised as follows. In section 2 we summarise the main features of the KGP model and give some of the technical details underlying it. In sections 3–5 we provide the computational models of some components of the KGP model and state their soundness wrt their formal specification. The overall computational model is built bottom-up, mirroring the hierarchical and modular structure of the abstract model. Section 3 also gives some background on the CL techniques that we have employed to define the KGP model, namely Abductive Logic Programming (ALP) and Logic Programming with Priorities (LPP), as well as the proof procedures (for ALP and for LPP) from which we have built the computational counterpart of the KGP model, in a bottom-up fashion. In section 6 we describe the prototype implementation of KGP agents, namely the SOCS-iC (for SOCS individual Computee) component of the PROSOCS platform [36]. Section 7 concludes.

2 KGP Model: Recap

Here we briefly summarise the KGP model for computees, see [18, 17] for any additional details. This model relies upon

- an internal (or mental) state,
- a set of reasoning capabilities, supporting planning, temporal reasoning, identification of preconditions of actions, reactivity and goal decision,
- a sensing capability,
- a set of transition rules, defining how the state of the computee changes, and defined in terms of the above capabilities,
- a set of selection functions, to provide appropriate inputs to the transitions,
- a cycle theory, for deciding which transitions should be applied when, and defined using the selection functions.

The model is defined in a modular and hierarchical fashion.

Internal State. This is a tuple $\langle KB, Goals, Plan, TCS \rangle$, where:

- KB is the knowledge base of the computee, and describes what the computee knows (or believes) of itself and of the environment. KB consists of modules supporting different reasoning capabilities:
  - KB_{plan}, for Planning,
  - KB_{pre}, for the Identification of Preconditions of actions,
  - KB_{TR}, for Temporal Reasoning,
  - KB_{GD}, for Goal Decision,
  - KB_{react}, for Reactivity, and
  - KB_0, for holding the (dynamic) knowledge of the computee about the external world in which it is situated (including past communications).

Syntactically, KB_{plan},KB_{react} and KB_{TR} are abductive logic programs with constraint predicates (see section 3.1), KB_{pre} is a logic program (see section 3.1), KB_{GD} is a logic program with priorities (see section 3.2), and KB_0 is a set of logic programming facts, and it is (implicitly) included in all the other modules.

– *Goals* is the set of properties that the computee wants to achieve, each one explicitly time-stamped by a time variable. Goals may also be equipped with a temporal constraint (belonging to TCS) bounding the time variable and defining when the goals are expected to hold. Goals may be *mental* or *sensing*. Both can be observed to hold (or not to hold) via the Sensing capability. In addition, mental goals can be brought about actively by the computee by its Planning capability and its actions.

– *Plan* is a set of actions scheduled in order to satisfy goals. Each is explicitly time-stamped by a time variable and possibly equipped with a temporal constraint, similarly to *Goals*, but defining when the action should be executed. Actions are partially ordered, via their temporal constraints. Each action is also equipped with the preconditions for its successful execution, determined by the Identification of Preconditions capability. Actions may be *physical*, *communicative*, or *sensing*. We assume that actions are atomic and do not have a duration. Actions can be seen as special kinds of goals which are directly executable.

– TCS is a set of constraint atoms (referred to as *temporal constraints*) in some given underlying constraint language with respect to some structure $\Re$ equipped with a notion of constraint satisfaction $\models_{\Re}$(see section 3.1). We assume that the constraint predicates include $<, \leq, >, \leq, =, \neq$. Temporal constraints refer to time constants, namely numbers, and time variables, namely distinguished variables which can be instantiated to time constants. These constraints bind the time of goals in *Goals* and actions in *Plan*. For example, they may specify a time window over which the time of an action can be instantiated, at execution time.

Goals and actions are uniquely identified by their associated time variable, which is implicitly existentially quantified within the overall state.

To aid revision and partial planning, *Goals* and *Plan* form a *tree*[2]. The tree is augmented by calls to the Goal Decision, Planning and Reactivity capabilities. The tree is given implicitly by associating with each goal and action its parent. *Top-level* goals and actions are children of the root of the tree, which, by convention, is the special symbol $\perp$. Actions always occur as leaves.

[2] In the full model [5], we actually have two trees, the first containing *non-reactive* goals and actions, the second containing *reactive* goals and actions. All the top-level non-reactive goals are either assigned to the computee by its designer at birth, or they are determined by the Goal Decision capability. All the top-level reactive goals and actions are determined by the Reactivity capability. Here for simplicity we overlook the distinction amongst the two trees.

Reasoning Capabilities. These are:

- Planning, which generates partial plans for sets of goals. It provides (temporally constrained) sub-goals and actions designed for achieving the input goals.
- Reactivity, which reacts to perceived changes in the environment, by replacing (some) goals in *Goals* and actions in *Plan* with (possibly temporally constrained) goals and actions.
- Goal Decision, which revises the top-most level goals of the computee, adapting the computee's state to changes in its own preferences and in the environment. Differently form Reactivity, it only modifies the top-level goals, it does not add actions to *Plan* and it does not depend upon the current *Goals* and *Plan*.
- Identification of Preconditions for action execution.
- Temporal Reasoning, which reasons about the evolving environment, and makes predictions about properties (fluents) holding in the environment, based on the partial information the computee acquires.

Sensing Capability. In addition to the reasoning capabilities above, the computee is equipped with a Sensing capability which links it to its environment, by allowing to observe that properties hold or do not hold, and that other agents have executed actions in the past.

Transitions. The state of a computee evolves by applying transition rules, which employ capabilities and the constraint satisfaction $\models_{\Re}$. The transitions are:

- Goal Introduction (GI), changing the top-level *Goals*, and using Goal Decision.
- Plan Introduction (PI), changing *Goals* and *Plan*, and using Planning and Introduction of Preconditions.
- Reactivity (RE), changing *Goals* and *Plan*, and using the Reactivity capability.
- Sensing Introduction (SI), changing *Plan* by introducing new sensing actions for checking the preconditions of actions already in *Plan*, and using Sensing.
- Passive Observation Introduction (POI), changing KB_0 by introducing unsolicited information coming from the environment, and using Sensing.
- Active Observation Introduction (AOI), changing KB_0 by introducing the outcome of (actively sought) sensing actions, and using Sensing.
- Action Execution (AE), executing all types of actions, and thus changing KB_0.
- Goal Revision (GR), revising *Goals*, and using Temporal Reasoning and Constraint Satisfaction.
- Plan Revision (PR), revising *Plan*, and using Constraint Satisfaction.

Cycle. The behaviour of a computee is given by the application of transitions in sequences, repeatedly changing the state of the computee. These sequences are not determined by fixed cycles of behaviour, as in conventional agent architectures, but rather by reasoning with cycle theories. These are logic programs with priorities (see section 3.2), defining preference policies over the order of application of transitions, which may depend on the environment and the internal state of a computee. This provision of a declarative control for computees in the form of cycle theories is a highly novel feature of the model, which could, in principle, be imported into other agent systems.

In the remainder of this section, we give some details on the state of computees. Details on the other components of the model (capabilities, transitions, selection functions and cycle) will be illustrated when describing their computational models, in sections 3–5. Here, note that alternative choices for reasoning capabilities and transitions would have been possible, and the model could be extended to incorporate further such capabilities and further transitions.

Vocabularies. We assume (possibly infinite) vocabularies of *time constants* (e.g., the set of all natural numbers), *time variables* (indicated with $t, t', s, \ldots$), *fluents* (indicated with $f, f', \ldots$), *action operators* (indicated with $a, a', \ldots$), and *names of computees* (indicated with c, c', ...). Given a fluent f, f and $\neg f$ are referred to as *fluent literals*. We use $l, l', \ldots$ to denote fluent literals. Moreover, given a fluent literal l, by $\bar{l}$ we denote its complement, namely $\neg f$ if l is f, f if l is $\neg f$.

We assume that the set of fluents is partitioned in two disjoint sets: *mental fluents* and *sensing fluents*. Intuitively, mental fluents represent properties that the computee itself is able to plan for so that they can be satisfied, but can also be observed. On the other hand, sensing fluents represent properties which are not under the control of the computee and can only be observed by sensing the external environment. For example, $problem\_fixed$ and $get\_resource$ may represent mental fluents, namely the properties that (given) problems be fixed and (given) resources be obtained, whereas $request\_accepted$ and $connection\_on$ may represent sensing fluents, namely the properties that a request for some (given) resource is accepted and that some (given) connection is active.

We also assume that the set of action operators is partitioned into three disjoint sets: *sensing*, *physical*, and *communication action operators*. Intuitively, sensing actions represent actions that the computee performs in order to establish whether some fluents hold in the environment. These fluents may be *sensing* fluents, but they can also represent effects of actions that the computee may need to check in the environment. On the other hand, *physical* actions are actions that the computee performs in order to achieve some specific effect, which typically causes some changes in the environment. Finally, *communication* actions are actions which involve communications with other computees. For example, $sense(connection\_on, t)$ is a sensing action, aiming at checking whether or not the sensing fluent $connection\_on$ holds; $do(clear\_table, t)$ may be a physical action operator, and $tell(c_1, c_2, request(r1), d, t)$ may be a communication action expressing that computee c_1 is requesting from computee c_2 the resource r_1 within a dialogue with identifier d, at time t.

Goals. A goal G is a pair of the form $\langle l[t], G' \rangle$ where

- $l[t]$ is the *fluent literal* of the goal, referring to a time variable t;
- G' is the *parent* of G.

Top-level goals are goals of the form $G = \langle l[t], \bot \rangle$. As an example, we may have a top-level goal G of the form $\langle problem\_fixed(p2, t), \bot \rangle$ and a subgoal G' of G of the form $\langle get\_resource(r1, t'), G \rangle$, with $TCS = \{5 \leq t \leq 10, 5 \leq t' < t\}$, meaning that to fix problem $p2$ within a certain time interval, the computee needs to have (or acquire) a resource $r1$ within an appropriate other time interval.

Mental (sensing) goals are goals whose fluent is mental (sensing, respectively).

Actions. An action A is a triple of the form $\langle a[t], G, C \rangle$ where

- $a[t]$ is the *operator* of the action, referring to the execution time variable t;
- G is the goal towards which the action contributes (i.e., the action belongs to a plan for the goal G). G may be a post-condition for A (but there may be other such post-conditions).
- C are the *preconditions* which should hold in order for the action to take place successfully; syntactically, C is a conjunction of (timed) fluent literals.

As an example, we may have an action $\langle tell(c_1, c_2, request(r1), d, , t''), G', \{\} \rangle$ within the state of some computee c_1, where G' is given above, $5 \leq t'' < t'$ also belongs to TCS, and c_2 is the name of some other computee.

(Non-)Sensing actions are actions whose operator is a (non-)sensing one.

Time Variables. In both a timed fluent literal $l[t]$ and a timed operator $a[t]$, the time t is a time variable. This variable is treated as an existentially quantified variable, the scope of which is the whole *state* of the computee. Whenever a goal (respectively action) is introduced within a state, the time variable associated with the goal (respectively action) is to be understood as a distinguished, fresh variable, serving as its identifier. When a time variable is instantiated (e.g., at action execution time) the actual instantiation is recorded in (the KB_0 part of) the state of the computee. This allows us to keep different instances of the same action (respectively goal) distinguished.

For simplicity, we assume that, given a state $\langle KB, Goals, Plan, TCS \rangle$, all occurrences of variables in $Goals$ and $Plan$ are time variables. In other words, our goals and actions are ground except for the time parameter. Variables other than time variables in goals and actions can be dealt with similarly. We concentrate on time variables as time plays a fundamental role in our model, and we avoid dealing with the other variables to keep the presentation of the model simple.

KB_0. Amongst the various modules in KB, we distinguish KB_0, which records the actions which have been executed (by the computee or by others) and their time of execution as well as the properties (i.e. fluents and their negation) which have been observed and the time of the observation. Formally, KB_0 contains assertions of the form:

- $executed(a[t], \tau)$ where $a[t]$ is a timed operator and τ is a time constant, meaning that action a has been executed at time $t = \tau$ by the computee.
- $observed(l[t], \tau)$ where $l[t]$ is a timed fluent literal and τ is a time constant, meaning that the property l has been observed to hold at time $t = \tau$.

- *observed*$(c, a[\tau'], \tau)$ where c is a computee's name, different from the name of the computee whose state we are defining, τ and τ' are time constants, and a is an action operator. This means that the given computee has observed at time τ that computee c has executed the action a at time τ' ($\tau' \leq \tau$).

Note that assertions in KB_0 of the third kind are variable-free. These are intended, e.g., to represent reception of communication from other computees. Instead, assertions of the first two kinds refer explicitly to a time variable t. This representation with explicit variables allows us to instantiate implicitly the time variable of (executed) actions and (observed) goals, via $\Sigma(S)$ (see below), while keeping the time variable explicitly as an identifier for actions and goals. As a consequence, the time variables in KB_0 are not properly speaking variables as such.

Since KB_0 is used in all the remaining modules in KB, and these are represented in a logic programming style, we are not allowed to have assertions with existentially quantified variables. Hence, the various knowledge bases will include a *variant* of KB_0, namely $KB_0\Sigma(S)$, where $\Sigma(S)$ is defined below. We will refer to $KB_0\Sigma(S)$ simply as KB_0.

Valuation of Time Variables and Temporal Constraints.
Given a state $S=\langle KB, Goals, Plan, TCS \rangle$, we denote by $\Sigma(S)$ (or simply Σ, when S is clear from the context) the valuation:

$$\Sigma(S) = \{t = \tau \mid executed(a[t], \tau) \in KB_0\} \cup \{t = \tau \mid observed(l[t], \tau) \in KB_0\}$$

Intuitively, Σ extracts from KB_0 the instantiation of the (existentially quantified) time variables in *Plan* and *Goals*, derived from having executed (some of the) actions in *Plan* and having observed that (some of the) fluents in *Goals* hold (or do not hold). Thus, KB_0 provides a "virtual" representation of Σ.

Below, $\Sigma(t)$, for some time variable t, will return the value of t in Σ, if there exists one, namely, if $t = \tau \in \Sigma$, then $\Sigma(t) = \tau$. The valuation of any temporal constraint Tc in a state S will always take Σ into account. Namely, any ground valuation for the temporal variables in Tc must agree with Σ on the temporal variables assigned to in Σ.

3 Computational Models for Capabilities

The reasoning capabilities of Planning, Reactivity, Identification of Preconditions and Temporal Reasoning are specified within the framework of Abductive Logic Programming (ALP), and the reasoning capability of Goal Decision is specified within the framework of Logic Programming with Priorities (LPP). Their computational models rely upon proof procedures for ALP and LPP (as appropriate). In this section, we briefly recall ALP and LPP, and summarise the concrete proof procedures used for the computational model of the reasoning capabilities. Finally, we give detailed specification and computational model for Planning (chosen as the representative ALP-based capability) and for Goal

Decision. The remaining capabilities, of Identification of Preconditions and Reactivity, are briefly mentioned. We also state the soundness results for these capabilities, building upon the soundness results for the underlying procedures. For details and proofs see [5].

3.1 ALP-Based Capabilities

Background: Abductive Logic Programming with Constraints. An *abductive logic program* with constraints is a tuple $\langle \Re, P, A, I \rangle$ where:

- $\Re$ is a structure consisting of a domain $D(\Re)$ and a set of constraint predicates including equality, together with an assignment of relations on $D(\Re)$ for each constraint predicate. The structure is equipped with a notion of $\Re$-satisfiability. Given (a set of) constraints C, $\models_{\Re} C$ stands for C is $\Re$-satisfiable, and $\sigma \models_{\Re} C$, for some grounding σ of the variables of C over $D(\Re)$, stands for C is $\Re$-satisfied by σ.
- P is a *normal logic program with constraints*, namely a set of rules of the form $H \leftarrow L_1 \wedge \ldots \wedge L_n$ with H atom, L_i literals, and $n \geq 0$. Literals can be positive, namely atoms, or negative, namely of the form *not* B, where B is an atom, or constraint atoms over $\Re$. The negation symbol *not* indicates *negation as failure* [8]. All variables in H, L_i are implicitly universally quantified, with scope the entire rule. If $n = 0$, the rule is called a *fact*.
- A is a set of *abducible predicates* in the language of P. Atoms whose predicate is abducible are referred to as *abducible atoms* or simply as *abducibles*.
- I is a set of *integrity constraints*, that is, a set of sentences in the language of P. All the integrity constraints in the KGP model have the implicative form $L_1 \wedge \ldots \wedge L_n \Rightarrow A_1 \vee \ldots \vee A_m \quad (n \geq 0, m > 0)$ where L_i are literals (as in the case of rules)[3], A_j are atoms (possibly the special atom $false$). All variables in the integrity constraints are implicitly universally quantified from the outside.

Given an abductive logic program $\langle \Re, P, A, I \rangle$ and a formula (*query*) Q, which is an (implicitly existentially quantified) conjunction of literals in the language of the abductive logic program, the purpose of abduction is to find a (possibly minimal) set of (ground) abducible atoms Γ which, together with P, "entails" (an appropriate ground instantiation of) Q, with respect to some notion of "entailment" that the language of P is equipped with, and such that the extension of P "satisfies" I (see [19] for possible notions of integrity constraint "satisfaction"). Here, the notion of "entailment" depends on the semantics associated with the logic program P (there are many different possible choices for this [19]), appropriately combined with the notion of $\Re-$satisfiability, as in Constraint Logic Programming [16]. We will refer to such a combined semantics as $\models_{LP(\Re)}$.

Formally and concretely, given a query Q, a set Δ of (possibly non-ground) abducible atoms, and a set C of (possibly non-ground) constraints, the pair (Δ, C) is an *abductive answer (with constraints)* for Q, with respect to an abductive logic program with constraints $\langle \Re, P, A, I \rangle$, iff for all groundings σ for

[3] If $n = 0$, then $L_1, \ldots, L_n$ represents the special atom *true*.

the variables in Q, Δ, C such that $\sigma \models_{\Re} C$, it holds that (i) $P \cup \Delta\sigma \models_{LP(\Re)} Q\sigma$, and (ii) $P \cup \Delta\sigma \models_{LP(\Re)} I$. Here, $\Delta\sigma$ plays the role of Γ in the earlier informal description of abductive answer.

Such notion can be extended to take into account an initial set of (possibly non-ground) abducible atoms Δ_0 and an initial set of (possibly non-ground) constraint atoms C_0, so that an abductive answer for Q, with respect to $\langle \Re, P, A, I \rangle$, Δ_0, C_0, is a pair (Δ, C) such that $\Delta \cap \Delta_0 = \{\}$, $C \cap C_0 = \{\}$, and $(\Delta \cup \Delta_0, C \cup C_0)$ is an abductive answer for Q, with respect to $\langle \Re, P, A, I \rangle$ (in the earlier sense).

In the sequel, for simplicitly, we will omit $\Re$ from abductive logic programs.

In ALP (with constraints), abductive answers are computed via *abductive proof procedures*, which typically extend SLD-resolution, providing the computational backbone underneath most logic programming systems, in order to check and enforce integrity constraint satisfaction, the generation of abducible atoms, and the satisfiability of constraint atoms (if any). There are a number of such procedures in the literature, e.g. the A-system [24]. To provide a computational counterpart to (the abductive tasks in) the KGP model, we propose and adopt the CIFF proof procedure [11, 12], extending the IFF proof procedure [13] for the purposes of the SOCS project. This procedure is summarised next. Full details are given in [11].

CIFF: A Proof Procedure for ALP with Constraints. CIFF extends IFF by dealing with constraints and non-allowed abductive logic programs, by tackling the issue of allowedness *dynamically*, i.e. at runtime, rather than adopting a static and overly strict set of allowedness conditions as in IFF [13]. To this end, the CIFF procedure includes a *dynamic allowedness rule* which is triggered whenever the procedure encounters a particular formula it cannot manipulate correctly due to a problematic quantification pattern.

In defining CIFF, we assume the availability of a sound and complete constraint solver, that we use as a *black box* component of the procedure. We do not make any assumption on the language of constraints, except for assuming that it includes a relation symbol for equality and it is closed under complements.

Input. Given an abductive logic program $\langle P, A, I \rangle$, the *input* to the CIFF procedure consists of a *query* Q, the set of *integrity constraints* I, and, in the background, a *theory* Th, which is a set of iff-definitions obtained by completing [8] the non-abducible, non-constraint predicates in the language of $\langle P, A, I \rangle$. Thus, the set of *abducibles* A is implicitly the set of predicates for which there is no definition in Th. The iff-definitions in Th have the following form:

$$p(X_1, \ldots, X_k) \leftrightarrow D_1 \vee \cdots \vee D_n$$

Negative literals are treated as implications (e.g. $not\, q(X,Y)$ is treated as $q(X,Y) \Rightarrow false$).

Output. There are three possible *outputs* of the CIFF procedure: (1) the procedure succeeds and produces an abductive answer to the query Q; (2) the procedure fails, thereby indicating that there is no abductive answer to the query

Q; and (3) the procedure reports that computing an abductive answer for the query Q is not possible, because a critical part of the input is not allowed.

Proof Rules. The CIFF procedure generates outputs from inputs by repeatedly applying a number of proof rules to the input. During this process, CIFF manipulates, essentially, formulas that are either atoms or implications or disjunctions of atoms and implications. Such formulas are referred to individually as *goals*. *Implications* are obtained by manipulating integrity constraints and (the rewriting of) negative literals. The theory *Th* is kept in the background and is only used to *unfold* defined predicates as they are being selected. Disjunction may thus be generated, and *splitting* will be applied to give rise to different branches in the proof search tree. The root of this tree is the original input $(Q \wedge I)$. *Nodes* of this tree are sets (conjunctions) of goals. CIFF repeatedly manipulates a selected node, via the proof rules, by rewriting goals in the node, adding new goals to the node, deleting superfluous goals from it, or adding $false$ to the node (and thus effectively deleting the node completely). For a full description of the proof rules, see [11]. Here, we only mention:

- *Case analysis for constraints:* Replace any goal of the form $Con \wedge A \Rightarrow B$, where Con is a constraint not containing any universally quantified variables, by $[Con \wedge (A \Rightarrow B)] \vee \overline{Con}$. There is a similar case analysis rule for equalities.
- *Constraint solving:* Replace any node containing an unsatisfiable set of constraints (as atoms) by $false$.
- *Dynamic allowedness rule:* Label nodes with problematic quantification patterns as *undefined*.

A node containing $false$ is called a *failure node*. If all leaf nodes in a search proof tree are failure nodes, then the derivation leading to that tree is said to be *failed* (the intuition being that there exists no answer to the query in question). A node to which no more proof rules can be applied is called a *final node*. A final node that is not a failure node and which has not been labelled as *undefined* is called a *success node*.

Answer Extraction. An *extracted answer* from a final success node N is a pair $\langle \Delta, C \rangle$, where Δ is the set of abducible atoms in N and C is obtained from the set of constraint atoms, equalities and disequalities in N. Below, we will use the following notations:

- $\langle P, A, I \rangle, Q \vdash_{CIFF} (\Delta, C)$ to stand for (Δ, C) is the answer extracted from a final success node obtained from the initial goal $Q \wedge I$;
- $\langle P, A, I \rangle, Q, \Delta_0, C_0 \vdash_{CIFF} (\Delta, C)$ to stand for $\langle P, A, I \rangle, Q \wedge \Delta_0 \wedge C_0 \vdash_{CIFF} (\Delta \cup \Delta_0, C \cup C_0)$ and $\Delta \cap \Delta_0 = \{\}$ and $C \cap C_0 = \{\}$;
- $\langle P, A, I \rangle, Q \vdash_{CIFF} fail$ to stand for: there is a failed derivation for Q;
- $\langle P, A, I \rangle, Q \vdash_{CIFF} flounder$ to stand for: there is a search proof tree for Q with no success leaf nodes and at least one undefined leaf node.

Soundness Results. We have shown that CIFF is *sound* [11, 12], and in particular the following result:

Theorem 1. (CIFF Soundness of Success) *Given a query Q and initial Δ_0, C_0:*
if $\langle P, A, I\rangle, Q, \Delta_0, C_0 \vdash_{CIFF} (\Delta, C)$
then (Δ, C) is an abductive answer for Q, with respect to $\langle P, A, I\rangle, \Delta_0, C_0$.

Specification and Computational Model for the ALP-Based Capabilities. KB_{plan}, KB_{react}, KB_{TR}, and KB_{pre} are all specified within the framework of the event calculus (EC) for reasoning about actions, events and changes [28]. Below, we give the abductive logic program KB_{plan} and the logic program KB_{pre}. KB_{react} is an extension of KB_{plan}, incorporating additional integrity constraints representing reactive rules. KB_{TR} is another variant of the EC, sharing a common kernel with KB_{plan}. Both KB_{react} and KB_{TR} are fully described in [5]. KB_{TR} is also given in [6].

Abductive Event Calculus for KB_{plan} and KB_{pre}. In a nutshell, the EC allows to write meta-logic programs which "talk" about object-level concepts of *fluents*, *events* (that we interpret as action *operations*), and *time points*. The main meta-predicates of the formalism are: $holds\_at(F, T)$ (a fluent F holds at a time T), $clipped(T_1, F, T_2)$ (a fluent F is clipped - from holding to not holding - between times T_1 and T_2), $declipped(T_1, F, T_2)$ (a fluent F is declipped - from not holding to holding - between times T_1 and T_2), $initially(F)$ (a fluent F holds from the initial time, say time 0), $happens(O, T)$ (an operation O happens at a time T), $initiates(O, T, F)$ (a fluent F starts to hold after an operation O at time T) and $terminates(O, T, F)$ (a fluent F ceases to hold after an operation O at time T). Roughly speaking, in a planning setting the last two predicates represent the cause-effects links between operations and fluents in the modelled world. We will also use a meta-predicate $precondition(O, F)$ (the fluent F is one of the preconditions for the executability of the operation O).

The EC allows to represent a wide variety of phenomena, including operations with indirect effects, non-deterministic operations, and concurrent operations [33]. A number of abductive variants of the EC have been proposed to deal with planning problems, e.g. see [32]. Here, we propose a novel variant, somewhat inspired by the $\mathcal{E}$-language [21], to allow situated agents to generate partial plans in a dynamic environment.

We give $KB_{plan} = \langle P_{plan}, A_{plan}, I_{plan}\rangle$. P_{plan} consists of two parts: domain-independent rules and domain-dependent rules. The basic *domain-independent rules*, directly borrowed from the original EC, are:

$$
\begin{array}{ll}
holds\_at(F, T_2) \leftarrow & happens(O, T_1), initiates(O, T_1, F), \\
 & T_1 < T_2, not\, clipped(T_1, F, T_2) \\
holds\_at(\neg F, T_2) \leftarrow & happens(O, T_1), terminates(O, T_1, F), \\
 & T_1 < T_2, not\, declipped(T_1, F, T_2) \\
holds\_at(F, T) \leftarrow & initially(F), 0 \leq T, not\, clipped(0, F, T) \\
holds\_at(\neg F, T) \leftarrow & initially(\neg F), 0 \leq T, not\, declipped(0, F, T) \\
clipped(T_1, F, T_2) \leftarrow & happens(O, T), terminates(O, T, F), T_1 \leq T < T_2 \\
declipped(T_1, F, T_2) \leftarrow & happens(O, T), initiates(O, T, F), T_1 \leq T < T_2
\end{array}
$$

The *domain-dependent rules* define *initiates*, *terminates*, and *initially*, e.g.

$$\begin{array}{ll} initiates(go(X,L_1,L_2),T,at(X,L_2)) \leftarrow & holds\_at(mobile(X),T) \\ initiates(go(X,L_1,L_2),T,free(L_1)) \leftarrow & holds\_at(mobile(X),T) \\ terminates(go(X,L_1,L_2),T,at(X,L_1) \leftarrow & holds\_at(mobile(X),T), L_1 \neq L_2 \\ terminates(go(X,L_1,L_2),T,free(L_2) \leftarrow & holds\_at(mobile(X),T), L_1 \neq L_2 \\ initially(at(bob,(1,1))) & \end{array}$$

Namely, the operation *go* from one location L_1 to some other location L_2 initiates the agent (robot) X being at location L_2 and location L_1 being free and terminates X being at location L_1 and location L_2 being free, provided that X is mobile. Moreover, some agent *bob* is initially at location $(1,1)$. The conditions for the rules defining *initiates* and *terminates* can be seen as preconditions for the effects of the operator *go* to take place. Preconditions for the executability of operators are specified within KB_{pre}, which contains a set of rules defining the predicate *precondition*, e.g.

$$\begin{array}{l} precondition(go(X,L_1,L_2),at(X,L_1)) \\ precondition(go(X,L_1,L_2),free(L_2)) \end{array}$$

namely the preconditions of the operator $go(X,L_1,L_2)$ are that X is at the initial location L_1 and that location L_2 X is moving to is free.

In order to accommodate (partial) planning we will assume that the domain-independent part in P_{plan} also contains the rules:

$$\begin{array}{l} happens(O,T) \leftarrow assume\_happens(O,T) \\ holds\_at(F,T) \leftarrow assume\_holds(F,T1), T_1 \leq T_2, not\ \ clipped(T_1,F,T_2) \\ holds\_at(\neg F,T) \leftarrow assume\_holds(\neg F,T1), T_1 \leq T_2, not\ \ declipped(T_1,F,T_2) \end{array}$$

i.e. an operator can be made to happen and a fluent can be made to hold simply by assuming them, where *assume_happens* and *assume_holds* are the only predicates in A_{plan} in KB_{plan}. This supports partial planning as follows. Actions $\langle a[t],\_,\_\rangle$ in the state amount to atoms $assume\_happens(a,t)$, thus, abducing an atom in the predicate *assume_happens* amounts to planning to execute the corresponding action. Moreover, goals $\langle l[t],\_\rangle$ in the state correspond to atoms $holds\_at(l,t)$ and $assume\_holds(l,t)$ (depending on whether they have already been planned for or not): thus, abducing atoms in the predicate *assume_holds* amounts to planning to further plan for the corresponding sub-goal.

I_{plan} contains the following domain-independent integrity constraints:

$$\begin{array}{l} holds\_at(F,T), holds\_at(\neg F,T) \Rightarrow false \\ assume\_happens(O,T), precondition(O,P) \Rightarrow holds\_at(P,T) \\ assume\_happens(O,T), not\ executed(O,T), time\_now(T') \Rightarrow T > T' \end{array}$$

namely a fluent and its negation cannot hold at the same time, when assuming (planning) that some action will happen, we need to enforce that each of its preconditions hold and that this action will be executable in the future.

To allow agents to draw conclusions from the contents of KB_0, which represent the “narrative” part of the computee’s knowledge, the following *bridge rules* are also amongst the domain independent rules of P_{plan}:

$$\begin{array}{ll}
clipped(T_1, F, T_2) \leftarrow & observed(\neg F, T), T_1 \leq T < T_2 \\
declipped(T_1, F, T_2) \leftarrow & observed(F, T), T_1 \leq T < T_2 \\
holds\_at(F, T_2) \leftarrow & observed(F, T_1), T_1 \leq T_2, not\ \ clipped(T_1, F, T_2) \\
holds\_at(\neg F, T_2) \leftarrow & observed(\neg F, T_1), T_1 \leq T_2, not\ \ declipped(T_1, F, T_2) \\
happens(O, T) \leftarrow & executed(O, T) \\
happens(O, T) \leftarrow & observed(C, O, T)
\end{array}$$

Note that we assume that the value of a fluent literal is changed according to observations only from the moment the observations are made, and actions by other agents have effects only from the time observations are made that they have been executed, rather than by the execution time itself. These choices are dictated by the rationale that observations can only be considered and reasoned upon from the moment the planning agent makes them.

Below, $KB^{\tau}_{plan} = \langle P_{plan} \cup \{time\_now(\tau)\}, A_{plan}, I_{plan}\rangle$.

Planning. The planning capability $\models^{\tau}_{plan}$ is specified as follows[4]. Let $S = \langle KB, Goals, Plan, TCS\rangle$ be a state, and $G = \langle l[t], \_\rangle$ be a mental goal in $Goals$. Let

– $\Delta_0 = \bigcup_{\langle a[t'],\_,\_\rangle \in Plan}\{assume\_happens(a, t')\} \cup$
 $\bigcup_{\langle l'[t'],\_\rangle \in Goals - \{G\}}\{assume\_holds(l', t')\}$
– $C_0 = TCS \wedge \Sigma(S)$.

Then, $S, G \models^{\tau}_{plan} (\mathcal{A}_s, \mathcal{G}_s, Tc)$ where

– $\mathcal{A}_s = \{a[t'] \mid assume\_happens(a, t') \in \Delta\}$ and
– $\mathcal{G}_s = \{l'[t'] \mid assume\_holds(l', t') \in \Delta\}$

for some (Δ, Tc) which is an abductive answer for $holds\_at(l, t)$, wrt KB^{τ}_{plan}, Δ_0 C_0. If no such abductive answer exists, then $S, G \models^{\tau}_{plan} \bot$, where $\bot$ is used here to indicate failure.

The computational counterpart of $\models^{\tau}_{plan}$, given by $\vdash^{\tau}_{plan}$, is defined in terms of CIFF, as follows.

– $S, G \vdash^{\tau}_{plan} (\mathcal{A}_s, \mathcal{G}_s, Tc)$ iff
  - $KB^{\tau}_{plan}, holds\_at(l, t), \Delta_0, C_0 \vdash_{CIFF} (\Delta, Tc)$
  - $\mathcal{A}_s = \{a[t] | assume\_happens(a, t) \in \Delta\}$
  - $\mathcal{G}_s = \{g[t] | assume\_holds(l, t) \in \Delta\}$.
– $S, G \vdash^{\tau}_{plan} \bot$ iff $KB^{\tau}_{plan}, holds\_at(l, t), \Delta_0, C_0 \vdash_{CIFF} X$ and $X = fail$ or $X = flounder$.

Directly from theorem 1, we prove soundness of $\vdash^{\tau}_{plan}$ wrt $\models^{\tau}_{plan}$, in the case of success.

Theorem 2. *(Planning Soundness of Success)*
If $\langle KB, Goals, Plan, TCS\rangle, G \vdash^{\tau}_{plan} (\mathcal{A}_s, \mathcal{G}_s, Tc)$,
then $\langle KB, Goals, Plan, TCS\rangle, G \models^{\tau}_{plan} (\mathcal{A}_s, \mathcal{G}_s, Tc)$.

[4] In the full model, we consider planning for multiple goals concurrently. Here, for simplicity we present the case of planning for single goals only.

Identification of Preconditions. This capability is specified as follows: given a timed action operator $a[t]$, $KB, a[t] \models_{pre} Cs$ iff

- either there exists c such that $KB_{pre} \models_{LP} precondition(a, c)$ and $Cs = \bigwedge\{c[t] \mid KB_{pre} \models_{LP} precondition(a, c)\}$
- or, otherwise, $Cs = true$.

The computational counterpart $\vdash_{pre}$ of this capability can be obtained by computing $\models_{LP}$ suitably. Trivially, if a sound and complete realisation of $\models_{LP}$ is used, the resulting $\vdash_{pre}$ is sound and complete with respect to $\models_{pre}$.

Reactivity. The specification of $\models^{\tau}_{react}$ and the provision of $\vdash^{\tau}_{react}$ are very similar to those of $\models^{\tau}_{plan}$ and $\vdash^{\tau}_{plan}$, and are omitted here for lack of space.

Temporal Reasoning. The temporal reasoning capability $\models_{TR}$ is invoked by other components of the KGP model (namely the Goal Decision capability, the Goal Revision transition and some of the selection functions, see section 4) to prove or disprove that a given (possibly temporally constrained) fluent literal holds (wrt the given theory KB_{TR}). We briefly summarise the specification of $\models_{TR}$ (see [5, 6] for the details).

- KB_{TR} is another variant of the EC, similar to KB_{plan}, and, analogously, is divided into a *domain-independent part*, a *domain dependent part*, and a *narrative part* KB_0, assumed not to contain "inconsistent" observations. The set of abducibles in KB_{TR} has *assume_holds* as its only abducible.
- $\models_{TR}$ is invoked by the Goal Decision capability to prove that a fluent literal $l[t]$, referring to a time constant t, holds wrt KB_{TR}, denoted as $KB \models_{TR} l[t]$. $\models_{TR}$ is invoked by the revision transitions and selection functions to prove that a fluent literal $l[t]$, referring to a time variable t constrained by some Tc, holds wrt KB_{TR}, denoted as $KB \models_{TR} l[t] \wedge Tc$.
- $\models_{TR}$ is understood *skeptically*, as follows. $KB \models_{TR} l[t] \wedge Tc$ (where Tc may be empty) iff (i) there exists an abductive answer for $holds\_at(l, t) \wedge Tc$, given KB_{TR}, and (ii) there exists no abductive answer for $holds\_at(\overline{l}, t) \wedge Tc$, given KB_{TR}, namely the fluent literal can be proven abductively, and its complement cannot.

The computational counterpart of $\models_{TR}$, given by $\vdash_{TR}$, is given by first providing a transformed version KB'_{TR} of KB_{TR}, and then by appropriate calls to the CIFF proof procedure, as follows (all details can be found in [6]).

- The transformation of KB_{TR} into KB'_{TR} relies upon the intuition that changes may happen only at significant time points, called *oases*, when events occur, while in the remaining time intervals, called *deserts*, nothing changes. Hence, it is possible to check for the validity of the query fluent literals and the integrity constraints in KB_{TR} with respect to oases only. The transformed KB'_{TR} addresses to some extent computational issues of viable realisation and scalability.

- $\vdash_{TR}$ is defined as follows, given $l[t] \wedge Tc$ (with Tc possibly empty):
  $KB \vdash_{TR} \; l[t] \wedge Tc$ iff
  - $KB'_{TR}, holds\_at(l, t) \wedge Tc \vdash_{CIFF} (\Delta, C)$, for some (Δ, C), and
  - $KB'_{TR}, holds\_at(\bar{l}, t) \wedge Tc \vdash_{CIFF} fail$.

  $KB \vdash_{TR}^{fail} l[t] \wedge Tc$ iff
  - $KB'_{TR}, holds\_at(l, t) \wedge Tc \vdash_{CIFF} fail$, or
  - $KB'_{TR}, holds\_at(l, t) \wedge Tc \vdash_{CIFF} (\Delta, C)$, for some (Δ, C), and
    $KB'_{TR}, holds\_at(\bar{l}, t) \wedge Tc \vdash_{CIFF} (\Delta', C')$, for some (Δ', C').

The following result of soundness directly follows from theorem 1 and from the equivalence between KB_{TR} and KB'_{TR}.

Theorem 3. *(Temporal Reasoning Soundness)*
Given $l[t] \wedge Tc$ (with Tc possibly empty):
If $KB_{TR} \vdash_{TR} \; l[t] \wedge Tc$ then $KB_{TR} \models_{TR} l[t] \wedge Tc$.
If $KB_{TR} \vdash_{TR}^{fail} l[t] \wedge Tc$ then $KB_{TR} \not\models_{TR} \; l[t] \wedge Tc$.

3.2 LPP-Based Capability: Goal Decision

Background: Logic Programming with Priorities. For the purposes of this paper, a *logic program with priorities*, referred to as $\mathcal{T}$, consists of four parts:

(i) a low-level part P, consisting of a logic program; each rule in P is assigned a name, which is a term; e.g., one such rule could be
$n(X) : p(X) \leftarrow q(X, Y), r(Y)$
with name $n(X)$;

(ii) a high-level part H, specifying conditional, dynamic priorities amongst rules in P; e.g., one such priority could be
$h(X) : m(X) \succ n(X) \leftarrow c(X)$
to be read: if (some instance of) the condition $c(X)$ holds, then (the corresponding instance of) the rule named by $m(X)$ should be given higher priority than (the corresponding instance of) the rule named by $n(X)$.

(iii) an auxiliary part A, defining predicates occurring in the conditions of rules in P, H and not in the conclusions of any rule in P;

(iv) a notion of incompatibility which, for our purposes, can be assumed to be given as a set of rules defining the predicate *incompatible*, e.g.
$incompatible(p(X), p'(X))$
to be read: any instance of the literal $p(X)$ is incompatible with the corresponding instance of the literal $p'(X)$. We assume that incompatibility is symmetric, and refer to the set of all incompatibility rules as I.

Any concrete LPP framework is equipped with a notion of entailment, that we denote by $\models_{pr}$, defined differently by different approaches to LPP, wrt some given underlying logic programming semantics $\models_{LP}$. Intuitively, $\mathcal{T} \models_{pr} \alpha$ iff α is the conclusion (wrt $\models_{LP}$) of a sub-theory of $P \cup A$ which is "preferred" wrt $H \cup A$ in $\mathcal{T}$ over any other sub-theory of $P \cup A$ that derives a conclusion incompatible with α (wrt I). For example, in [30, 25, 22], $\models_{pr}$ is defined via argumentation (see the next section).

The concrete framework for LPP that we adopt within the computational counterpart of the KGP model is that of Logic Programming without Negation as Failure ($LPwNF$) [9] suitably extended to deal with dynamic preferences [22]. Other concrete frameworks that could be used for LPP instead are, for instance, those presented in [30, 25].

***LPwNF*: An Argumentation-Based Framework for LPP.** In this section, we summarise the main features of $LPwNF$ and the notion of preference reasoning $\models_{pr}$, given with respect to an argumentation-based formulation of $LPwNF$.

$LPwNF$ is a concrete LPP framework, whereby the various components of a theory $\mathcal{T}$, as defined earlier, are as follows:

(i) The low-level part P consists of labelled propositional rules of the form $label : l \leftarrow l_1, ..., l_n$, where $l, l_1, ..., l_n$ are atoms a or explicitly negative literals $\neg a$. The underlying semantics, $\models_{LP}$, is given by the single inference rule of modus ponens. Non-ground rules are represented via all their ground instances in the given Herbrand universe of the program.
(ii) The high-level part H consists of propositional rules of the form $label : l \succ l' \leftarrow l_1, ..., l_n$, where $l_1, ..., l_n$ are atoms or explicitly negative literals and l, l' are labels of rules in P.
(iii) The auxiliary part A is a set of propositional rules of the form $l \leftarrow l_1, ..., l_n$.
(iv) The notion of incompatibility includes $incompatible(p, \neg p)$, for all atoms p, and $incompatible(r \succ s, s \succ r)$, for all labels of rules r and s.

We realise the notion of preference reasoning $\models_{pr}$ for $LPwNF$ through argumentation. Argumentation has recently been shown to be a useful framework for formalising non-monotonic reasoning and other forms of reasoning (see e.g. [4, 10, 20, 31, 30]). In general, an argumentation framework is a pair (Th, At) where Th is a theory in some background (monotonic) logic, equipped with an entailment $\models_{Th}$, and At is a binary *attacking relation* on the subsets of Th, i.e. $At \subseteq 2^{Th} \times 2^{Th}$. The subsets of Th form the arguments of the framework and At is therefore an attacking relation between arguments. We will write Δ *attacks* Δ' iff $(\Delta, \Delta') \in At$.

The semantics of an argumentation framework is based upon the following notion of *admissible* argument. An argument $\Delta \subseteq Th$ is admissible iff

- Δ does not attack itself,
- for all arguments $\Delta' \subseteq Th$, if Δ' attacks Δ, then Δ (counter-)attacks Δ'.

To provide $\models_{pr}$ for an $LPwNF$ theory $\mathcal{T}=(P, H, A, I)$, we view the latter as a concrete argumentation framework (Th, At) as follows. The set of arguments Th is given by $P \cup H \cup A$, with $\models_{Th}$ given by $\models_{LP}$. The attacking relation At is realised via a notion of *conflict* (using the notion of incompatibility I of $\mathcal{T}$) together with a notion of *strength* between arguments (using the preference rules in the H component of $\mathcal{T}$). Then, the preference semantics $\models_{pr}$ of an $LPwNF$ theory is given through argumentation in terms of the maximally admissible subsets of the corresponding argumentation framework. Usually, two variants of $\models_{pr}$ are defined. Given an $LPwNF$ theory $\mathcal{T}$, the corresponding argumentation framework (Th, At) and a formula F,

- $\mathcal{T} \models^{cred}_{pr} F$ iff there is one maximal (with respect to set inclusion) admissible argument Δ of (Th, At) where F holds, i.e. $\Delta \models_{LP} F$;
- $\mathcal{T} \models^{skep}_{pr} F$ iff $\mathcal{T} \models^{cred}_{pr} F$ and, for any G s.t. $I \cup A \models_{LP} \mathit{incompatible}(F, G)$, it holds that $\mathcal{T} \not\models^{cred}_{pr} G$.

In the KGP model, the preference entailment, $\models_{pr}$, is given by the skeptical entailment $\models^{skep}_{pr}$ in this section, when we define the Goal Decision capability, and by $\models^{cred}_{pr}$ in section 5, when we define the operational trace of computees via cycle theories.

Computing Preferential Reasoning in *LPwNF*. The computational counterpart of $\models_{pr}$ for $LPwNF$ is realised via a proof procedure, referred to as GORGIAS [5]. Given a formula F, GORGIAS aims to construct an admissible argument, Δ, that derives F, under the background logic $\models_{LP}$.

GORGIAS is based on an existing proof theory for computing admissible arguments for abstract argumentation frameworks [23]. This prooof theory is given in terms of derivations of trees, where nodes are arguments and each node is labelled as "attack" or "defence". A defence node is followed by a set of children attack nodes, one for each of its possible minimal (counte-)attacks. An attack node is followed by a defence node containing a (counter-)attack against its parent. *Successful derivations* terminate with a tree whose root, Δ, is an admissible argument supporting, via $\models_{LP}$, the initially given formula F. The root node Δ_0 of the initial tree is computed by reducing the given formula F into a minimal set that concludes F via $\models_{LP}$.

GORGIAS specialises this abstract proof theory by incorportating the $LPwNF$ specific way of computing attacks and counter-attacks. Any node N of the tree results from first choosing a "culprit" conclusion c of the parent node of N and then reducing (by resolution) some conflicting (incompatible) literal, $\overline{c}$, of c so that N minimally entails $\overline{c}$ under the background logic $\models_{LP}$.

The GORGIAS proof procedure then provides the following derivability relations for $LPwNF$. Given a theory $\mathcal{T}$ in $LPwNF$ and a literal L, let (Th, At) be the corresponding argumentation framework. Then

- $\mathcal{T} \vdash^{cred}_{pr} L$ iff there exists a successful derivation of GORGIAS for L.
- $\mathcal{T} \vdash^{sk}_{pr} L$ iff $\mathcal{T} \vdash^{cred}_{pr} L$ and $\mathcal{T} \not\vdash^{cred}_{pr} \overline{L}$ for any $\overline{L}$ such that $I \cup A \models_{LP}$ $\mathit{incompatible}(L, \overline{L})$.

We can then show that for finite theories of $LPwNF$, these derivability relations $\vdash^{sk}_{pr}$, $\vdash^{cred}_{pr}$ based upon GORGIAS, are sound and complete, as follows.

Theorem 4. *(soundness and completeness of $\vdash^{cred}_{pr}$ and $\vdash_{pr}$) Let $\mathcal{T}$ be a finite theory of LPwNF. Then the derivability relation $\vdash^{cred}_{pr}$ is sound and complete with respect to $\models^{cred}_{pr}$, provided that GORGIAS uses a sound and complete realisation of $\models_{LP}$. Hence the skeptical relation $\vdash^{sk}_{pr}$ is also sound and complete with respect to $\models^{skep}_{pr}$, provided that GORGIAS uses a sound and complete realisation of $\models_{LP}$.*

Specification and Computational Model for Goal Decision. The computee Goal Decision capability, $\models^{\tau}_{GD}$, selects, at a given instant, the top level goals to be pursued. These goals are preferred by the computee at the time of their selection, but this may change over time. This capability relies directly on the underlying preference reasoning within the $LPwNF$ framework. It simply uses this form of reasoning with a specific $LPwNF$ theory, KB_{GD}, in which the computee represents its goal preference policy.

The knowledge base KB_{GD} is written in four parts in the standard way as for any theory in $LPwNF$. The details of the specific form of its rules are ommitted due to lack of space. The main specialised forms of sentences in KB_{GD} are the following. Statements of incompatibility in KB_{GD}, $incompatible(l_1, l_2)$, are amongst literals, l_1 and l_2, referring to a subset of the fluents in the language of the computee separated out as the set of *goal fluents*. Rules in the basic (low-level) part of KB_{GD} have conclusions of the form $\langle g[t], Tg\rangle$ where g is a goal fluent, Tg is a (possibly empty) set of temporal constraints and the time variable t is existentially quantified with scope the conclusion of the rule. The auxiliary part, A, of KB_{GD} is augmented with $KB_0 \cup KB_{TR}$ and so the conditions of the rules in KB_{GD} are evaluated by combining the background derivability $\models_{LP}$ of the $LPwNF$ framework with the Temporal Reasoning capability $\models_{TR}$. Finally, any rule in KB_{GD} may have in its body a special atom, denoted by $now(\tau)$ that refers to the (current) time τ at which the capability of goal decision is applied by the computee. We will denote by KB^{τ}_{GD} the knowledge base obtained by adding to (the auxiliary part of) KB_{GD} the atom $now(\tau)$.

The capability of $\models^{\tau}_{GD}$ is defined directly in terms of the preference, $\models_{pr}$, of $LPwNF$ as follows. Given a state $\langle KB, Goals, Plan, TCS\rangle$, and a time point τ,

$$KB \models^{\tau}_{GD} \mathcal{G}s$$

iff $\mathcal{G}s$ is a maximal set, $\mathcal{G}s = \{\langle g_1[t_1], Tg_1\rangle, \ldots, \langle g_n[t_n], Tg_n\rangle\}$, $n \geq 0$, where g_i are goal fluent literals and Tg_i are temporal constraints on t_i, such that:

$$KB^{\tau}_{GD} \models_{pr} \langle g_1[t_1], Tg_1\rangle \wedge \ldots \wedge \langle g_n[t_n], Tg_n\rangle$$

This means that a new set of goals $\mathcal{G}s$ is generated that is currently (skeptically) preferred under the preference policy represented in KB_{GD} and the current information in KB_0, via the use of the Temporal Reasoning capability by $\models_{pr}$. Note that any two goals in $\mathcal{G}s$ are necessarily compatible with each other. Note also that $\models^{\tau}_{GD}$ may return an empty set of goals when there are no skeptically preferred goals at the time τ of application of this capability.

The derivability relation, $\vdash^{\tau}_{GD}$, and the computational model for Goal Decision can be drawn directly from the general GORGIAS proof procedure for $LPwNF$ and the derivability relations $\vdash^{cred}_{pr}$ and $\vdash_{pr}$ that this provides, as presented earlier. A simple but relatively inefficient way to compute $\vdash^{\tau}_{GD}$ would then be to generate one by one skeptical goals, via $\models_{pr}$, adding the most recently generated goal to the previous goals and re-checking, again via $\vdash_{pr}$, that the whole set remains a skeptical conclusion. A more efficient algorithm for computing $\vdash^{\tau}_{GD}$ that exploits some of the special features of KB_{GD} relies only on the credulous derivability relation of $LPwNF$ to *generate* in the first step a set of candidate goals and then, using checks of incompatibility, to *filter* from this the required goals.

Let us assume that the knowledge base, KB^{τ}_{GD}, for any given current time τ together with $KB_{TR} \cup KB_0$ added to the auxiliary part of KB_{GD} is such that only a finite number of goals can be derived from $\mathcal{T} = KB^{\tau}_{GD} \cup KB_{TR} \cup KB_0$ via the background logic $\models_{LP}$. We call this assumption the *goal finiteness* assumption. We also assume that the auxiliary part of $\mathcal{T}$ is consistent. Then, the following soundness result holds, directly from theorem 4.

Theorem 5. *(Goal Decision Soundness) Let $\mathcal{T} = KB^{\tau}_{GD} \cup KB_{TR} \cup KB_0$ have goal finiteness property and a consistent auxiliary part. Suppose that a sound and complete realisation of $\models_{LP}$ is used within $\vdash^{cred}_{pr}$. If $\mathcal{T} \vdash^{\tau}_{GD} Gs$ then $\mathcal{T} \models^{\tau}_{GD} Gs$.*

4 Computational Model for Transitions

The KGP model relies upon the state transitions GI, PI, RE, SI, POI, AOI, AE, GR, PR, as discussed in section 2. In [5], we have provided computational counterparts $\vdash_{GI}$, $\vdash_{PI}$, $\vdash_{RE}$, $\vdash_{SI}$, $\vdash_{POI}$, $\vdash_{AOI}$, $\vdash_{AE}$, $\vdash_{GR}$, $\vdash_{PR}$ for this transitions, defined via transition rules themselves, obtained from the specifications by replacing calls to capabilities appropriately by calls to their computational counterparts. Below, for one concrete transition (PI), we first summarise the formal specification, and then provide the computational countertpart.

Plan Introduction

This transition takes as input a state and a set of goals in the state (that have been selected by the *goal selection function*, see below) and produces a new state by calling the computee's Planning ($\models^{\tau}_{plan}$, see section 3.1) and Identification of Preconditions ($\models_{pre}$, see section 3.1) capabilities. For simplicity, we will provide specification and computational counterpart of this transition in the case of a single input goal (see [5] for the general case of multiple input goals).

Specification of (**PI**) $$\frac{\langle KB, Goals, Plan, TCS\rangle \qquad G}{\langle KB, Goals', Plan', TCS'\rangle}\,\tau$$

where G is a goal selected for planning and

$$\begin{aligned}&Goals' = Goals \cup Subg(G)\\&Plan' = Plan \cup Pplan(G)\\&TCS' = TCS \cup Tc\end{aligned}$$

where the sets $Subg(G)$, $Pplan(G)$ and Tc are obtained as follows.

(i) if G is a mental goal: let $\langle KB, Goals, Plan, TCS\rangle, G \models^{\tau}_{plan} X$. Then, either $X = \bot$ and $Subg(G) = Pplan(G) = Tc = \{\}$, or $X = (\mathcal{A}s, \mathcal{G}s, Tc)$ and $Subg(G) = \{\langle l[t], G\rangle \mid l[t] \in \mathcal{G}s\}$, $Pplan(G) = \{\langle a[t], G, P\rangle \mid a[t] \in \mathcal{A}s$ and $KB, a[t] \models_{pre} P\}$.

(ii) if $G = \langle l[t], G' \rangle$ is a sensing goal:
$Subg(G) = \{\}$, and
$Pplan(G) = \langle sense(l[t']), G', C \rangle$, where $KB_{pre}, sense(l[t']) \models_{pre} C$, and
$Tc = \{t' \leq t\}$.

Computational Counterpart of PI: $\vdash^{\tau}_{PI}$ consists of two components, dealing separately with mental and sensing goals. Below, given a partial plan X returned by $\vdash^{\tau}_{plan}$, we write $G(X)$ (respectively $A(X)$, $T(X)$), meaning $\{\}$ if $X = \bot$, and Gs (respectively As, Tc) if $X = (As, Gs, Tc)$.

$$\frac{\begin{array}{lcl} G & = & \langle l[t], \_ \rangle \quad \text{mental goal} \\ \langle KB, Goals, Plan, TCS \rangle, G \vdash^{\tau}_{plan} X & & \\ Goals_1 & = & Goals \cup G(X) \\ Plan_1 & = & Plan \cup A(X) \\ TCS_1 & = & TCS \cup T(X) \end{array}}{\langle KB, Goals, Plan, TCS \rangle, G \vdash^{\tau}_{PI} \langle KB, Goals_1, Plan_1, TCS_1 \rangle} (\vdash^{\tau}_{PI})$$

$$\frac{\begin{array}{lcl} G & = & \langle l[t], G' \rangle \quad \text{sensing goal} \\ KB_{pre}, sense(l[t']) \vdash_{pre} P & & \\ Plan_1 & = & Plan \cup \{\langle sense(l[t']), G', P \rangle\} \\ TCS_1 & = & TCS \cup \{t' \leq t\} \end{array}}{\langle KB, Goals, Plan, TCS \rangle, G \vdash^{\tau}_{PI} \langle KB, Goals, Plan_1, TCS_1 \rangle} (\vdash^{\tau}_{PI})$$

Clearly, if $\vdash^{\tau}_{plan}$ is correct wrt $\models^{\tau}_{plan}$ and $\vdash_{pre}$ is correct wrt $\models_{pre}$, then the computational counterpart $\vdash_{PI}$ is correct wrt PI. Thus, by theorem 2 (and the analogous theorem for $\models_{pre}$),

Theorem 6. *(Plan Introduction Soundness) Given states S, S', goal G, time-point τ, if $S, G \vdash^{\tau}_{PI} S'$ then* $(\mathbf{PI})\dfrac{S \quad G}{S'}\tau$.

Note that the PI transition (and its computational counterpart) relies upon a given input goal to be planned for. Such an input is provided within the KGP model via an appropriate *selection function* c_{GS}, defined in terms of capabilities and the constraint satisfaction $\models_{\Re}$. Similarly, the KGP model relies upon selection functions for action selection (to provide inputs to the AE transition), precondition selection (to provide inputs to the SI transition), and fluent selection (to provide inputs to the AOI transition). In the next section, we will see that these selection functions play another important role, of "enabling" transitions within cycles. In [5], we provide computational counterparts for all these selection functions, in terms of the computational counterparts of the various capabilities and the constraint satisfaction used therein.

5 Computational Model for Cycle

The role of the cycle theory is to dynamically control the sequence of the internal transitions that the agent applies in its "life". It regulates these "narratives of transitions" according to certain requirements that the designer of the agent would like to impose on the operation of the agent, but still allowing the possibility that any (or a number of) sequences of transitions can actually apply in the "life" of an agent. Thus, whereas a fixed cycle can be seen as a restrictive and rather inflexible catalogue of allowed sequences of transitions (possibly under pre-defined conditions), a cycle theory identifies *preferred patterns* of sequences of transitions. In this way a cycle theory regulates in a flexible way the operational behaviour of the agent.

Cycle theory. Formally, a cycle theory $\mathcal{T}_{cycle}$ consists of the following parts:

– An *initial* part $\mathcal{T}_{initial}$, that determines the possible transitions that the agent could perform when it starts to operate (*initial cycle step*). More concretely, $\mathcal{T}_{initial}$ consists of rules of the form

$$\mathcal{R}_{0|T}(S_0, X)\colon *T(S_0, X) \leftarrow C(S_0, \tau, X), now(\tau)$$

with name $\mathcal{R}_{0|T}(S_0, X)$, and sanctioning that, if the conditions C are satisfied in the initial state S_0 at the current time τ, then the initial transition should be T, applied to state S_0 and input X, if required. Note that the conditions C determine the input X of the first transition T. Such inputs are determined by calls to the appropriate selection functions (see section 4). Note also that $C(S_0, \tau, X)$ may be *true*, and $\mathcal{T}_{initial}$ might simply indicate a fixed initial transition T_1.

The notation $*T(S, X)$ in the head of these rules, meaning that the transition T can be potentially chosen as the next transition, is used in order to avoid confusion with the notation $T(S, X, S', \tau)$ that we have introduced earlier to represent the actual application of the transition T.

– A *basic* part $\mathcal{T}_{basic}$ that determines the transitions (*cycle steps*) that may follow other transitions, and consists of rules of the form

$$\mathcal{R}_{T|T'}(S', X')\colon *T'(S', X') \leftarrow T(S, X, S', \tau), EC(S', \tau', X'), now(\tau')$$

with name $\mathcal{R}_{T|T'}(S', X')$, and sanctioning that, after the transition T has been executed, starting at time τ in the state S and ending at the current time τ' in the resulting state S', and the conditions EC evaluated in S' at τ' are satisfied, then transition T' could be the next transition to be applied in the state S' with the (possibly empty) input X', if required. The conditions EC are called *enabling conditions* as they determine when a cycle-step from the transition T to the transition T' can be applied. In addition, they determine the input X' of the next transition T'. Such inputs are determined by calls to the appropriate selection functions.

– A *behaviour* part $\mathcal{T}_{behaviour}$ that contains rules describing dynamic priorities amongst rules in $\mathcal{T}_{basic}$ and $\mathcal{T}_{initial}$. Rules in $\mathcal{T}_{behaviour}$ are of the form

$$\mathcal{R}_{T|T'}(S, X') \succ \mathcal{R}_{T|T''}(S, X'') \leftarrow BC(S, X', X'', \tau), now(\tau)$$

with $T' \neq T''$, which we refer to via the name $\mathcal{P}^{T}_{T' \succ T''}$. Recall that $\mathcal{R}_{T|T'}(\cdot)$ and $\mathcal{R}_{T|T''}(\cdot)$ are (names of) rules in $\mathcal{T}_{basic} \cup \mathcal{T}_{initial}$. Note that, with an abuse of notation, T could be 0 in the case that one such rule is used to specify

a priority over the *first* transition to take place, in other words, when the priority is over rules in $\mathcal{T}_{initial}$. These rules in $\mathcal{T}_{behaviour}$ sanction that, at the current time τ, after transition T, if the conditions BC hold, then we prefer the next transition to be T' over T'', namely doing T' has *higher priority* than doing T'', after T. The conditions BC are called *behaviour conditions* and give the behavioural profile of the agent. These conditions depend on the state of the agent after T and on the parameters chosen in the two cycle steps represented by $\mathcal{R}_{T|T'}(S, X')$ and $\mathcal{R}_{T|T''}(S, X'')$. Behaviour conditions are *heuristic* conditions, which may be defined in terms of *heuristic selection functions* (see [17] for details). For example, the heuristic action selection function may choose those actions in the agent's plan whose time is close to running out amongst those whose time has not run out.
- An *auxiliary part* including definitions for any predicates occurring in the enabling and behaviour conditions, and in particular for selection functions (including the heuristic ones, if needed).
- An *incompatibility part*, including rules stating that all different transitions are incompatible with each other:

$$incompatible(*T(S, X), *T'(S, X'))$$

for all T, T' such that $T \neq T'$, and that different calls to the same transition but with different input items are incompatible with each other:

$$incompatible(*T(S, X), *T(S, X')) \leftarrow X \neq X'$$

Overall, these rules express that only one transition can be chosen at a time.

Hence, $\mathcal{T}_{cycle}$ is an LPP-theory (see Section 3.2) where:

(i) $P = \mathcal{T}_{initial} \cup \mathcal{T}_{basic}$, and
(ii) $H = \mathcal{T}_{behaviour}$.

In the sequel, we will indicate with $\mathcal{T}^0_{cycle}$ the sub-cycle theory $\mathcal{T}_{cycle} \setminus \mathcal{T}_{basic}$ and with $\mathcal{T}^s_{cycle}$ the sub-cycle theory $\mathcal{T}_{cycle} \setminus \mathcal{T}_{initial}$.

Operational Trace. A cycle theory $\mathcal{T}_{cycle}$ is used to induce *cycle operational traces* of an agent, namely a (typically infinite) sequence of transitions

$$T_1(S_0, X_1, S_1, \tau_1), \ldots, T_i(S_{i-1}, X_i, S_i, \tau_i),\ T_{i+1}(S_i, X_{i+1}, S_{i+1}, \tau_{i+1}), \ldots$$

(where each of the X_i may be empty), such that

- S_0 is the given initial state;
- for each $i \geq 1$, τ_i is given by the clock of the system, such that $\tau_i < \tau_{i+i}$;
- (*Initial Cycle Step*) $\mathcal{T}^0_{cycle} \wedge now(\tau_1) \models_{pr} *T_1(S_0, X_1)$;
- (*Cycle Step*) for each $i \geq 1$
$\mathcal{T}^s_{cycle} \wedge T_i(S_{i-1}, X_i, S_i, \tau_i) \wedge now(\tau_{i+1}) \models_{pr} *T_{i+1}(S_i, X_{i+1})$
namely each (non-final) transition in a sequence is followed by the most preferred transition, as specified by $\mathcal{T}^s_{cycle}$. If the most preferred transition determined by $\models_{pr}$ is not unique, we choose arbitrarily one.

Computational Counterpart of Operational Trace. Since the notion of operational trace is based upon $\models_{pr}$ and transitions (and selection functions), its computational counterpart is obtained by replacing $\models_{pr}$ with its computational counterpart $\vdash_{pr}$ and transitions with their computational counterparts. Thus, a computational operational trace is a (possibly infinite) sequence of computational counterparts of transitions, of the form

$$T_1^c(S_0, X_1, S_1, \tau_1), \ldots, T_i^c(S_{i-1}, X_i, S_i, \tau_i),\ T_{i+1}^c(S_i, X_{i+1}, S_{i+1}, \tau_{i+1}), \ldots$$

where

- $\mathcal{T}_{cycle}^0 \wedge now(\tau_1) \vdash_{pr} *T_1(S_0, X_1)$ and,
- for each $i \geq 1$, $\mathcal{T}_{cycle}^s \wedge T^i(S_{i-1}, X_i, S_i, \tau_i) \wedge now(\tau_{i+1}) \vdash_{pr} *T_{i+1}(S_i, X_{i+1})$.

Trivially, computational counterparts of operational traces correspond to operational traces, thanks to the soundness results for each transition (similarly to theorem 6) and the soundness of $\models_{pr}$ (see theorem 4).

6 Implementation

To realise the logical and computational aspects of the KGP model we have developed PROSOCS [36], a platform which allows us to deploy and test the functionality of KGP agents via the SOCSiC component of PROSOCS. Deployment of KGP agents using SOCSiC is based on an *agent template* whose design [35] builds upon previous work in multi–agent systems, in particular, the *head/body* metaphor described by [37] and [14], and the *mind/body* architecture introduced by [3] and more recently used by [15].

In the mind part of a PROSOCS agent, the ALP-based components of the KGP model are implemented in CIFF [11, 12] and the LPP-based components of the KGP model are implemented in GORGIAS [1]. Overall, we build the mind using SICStus Prolog [34] and the bidirectional Java-Prolog interface Jasper it provides; Jasper is used by the body to exchange information with the mind.

To implement the body of the agent we use Java *on top* of the Peer-to-Peer JXTA Project [39]. JXTA is suitable for the low-level functionality of a PROSOCS agent, such as interaction with the environment, and is provided in the form of an API (Application Programming Interface). By importing this API when we instantiate specific PROSOCS agents, we enable such agents to discover bodies of other PROSOCS agents (using JXTA's peer discovery protocols facilities for dynamic discovery in a GC network) as well as communicate with other agents (using JXTA's facilities for message transport and structuring via a pipe binding and resolver protocols).

To facilitate experimentation with KGP agents we have built interfaces, which allow us to animate an agent's behaviour while interacting with other agents in the context of a GC application. The aspects of the agent's behaviour that we animate, in the current state of the implementation, are: the computational trace in terms of the names of transitions being executed, the observations the agent makes and the actions it executes, the internal state of the agent in terms of its knowledge, goals and plans. More details of the implementation of the agent template can be found in [2, 36].

7 Conclusions

In this paper we have summarised the declarative model of the KGP agents [18] and then given its computational counterpart and briefly described its implementation. The declarative model is based on computational logic, in particular on abductive logic programming and logic programming with priorities. It is modular, hierarchical and extensible. It specifies a collection of capabilities, uses them to define a collection of transitions, to be used within logically specified context sensitive cycle theories. In close relationship to the declarative model, the computational model mirrors the logical architecture by specifying appropriate computational counterparts for the capabilities and using these to give the computational models of the transitions. These computational models and the one specified for the cycle theories are all based on, and are significant extensions of, existing proof procedures for abductive logic programming and logic programming with priorities. This design has satisfied two of the main motivations of the KGP model, namely to reduce the gap between the logical and computational realisations, and to exploit and integrate (extensions of) existing computational logic tools and techniques.

The paper also reports a first prototype implementation of the KGP model based on SICStus Prolog, Java and JXTA [36]. The implementation reflects the modular and hierarchical architecture and is similarly extensible. The prototype has been used successfully on a number of small scenaria demonstrating, amongst others, the situatedness of agents and their adaptability and responsiveness to changes in their environment, as required by the GC challenge.

The KGP model does not currently include a number of features, some of which are subject of future work. It would be advantageous, for example, to include a more sophisticated way to assimilate and revise knowledge, for example using inductive logic programming. Another useful feature would be to allow agents to have in their knowledge, and to reason with, what they believe other agents believe. Such knowledge can be used, for example, in communication strategies and even possibly in planning. A particular issue included in the GC vision and not addressed in the KGP model to date is physical mobility, although we do cater, at the knowledge level, for agents moving from one society to another [38]. This issue will necessitate further work. Finally, we are currently working on identifying and verifying formally properties of behaviour of computees.

Acknowledgements

This work was funded by the IST programme of the EC, FET under the IST-2001-32530 SOCS project, within the GC proactive initiative. K. Stathis and F. Toni were also supported by the Italian MIUR programme “Rientro dei Cervelli”.

References

1. *Gorgias User Guide:* `http://www.cs.ucy.ac.cy/~nkd/gorgias/`, 2003.
2. M. Alberti, A. Bracciali, F. Chesani, U. Endriss, M. Gavanelli, W. Lu, K. Stathis, and P. Torroni. SOCS prototype. Technical report, SOCS Consortium, 2003. Deliverable D9.
3. J. Bell. A Planning Theory of Practical Rationality. In *Proceedings of AAAI'95 Fall Symposium on Rational Agency*, pages 1–4. AAAI Press, 1995.
4. A. Bondarenko, P. M. Dung, R. A. Kowalski, and F. Toni. An abstract, argumentation-theoretic approach to default reasoning. *Artificial Intelligence*, 93:63–101, 1997.
5. A. Bracciali, N. Demetriou, U. Endriss, M. Gavanelli, A. C. Kakas, E. Lamma, P. Mancarella, P. Mello, P. Moraitis, F. Sadri, K. Stathis, G. Terreni, F. Toni, and P. Torroni. Computational model for computees and societies of computees. Technical report, SOCS Consortium, 2003. Deliverable D8.
6. A. Bracciali and A. Kakas. Frame consistency: Reasoning with explanations. In *Proceedings of the 10th International Workshop on "Non-Monotonic Reasoning" (NMR2004)*, Whistler BC, Canada, 2004.
7. F. Chesani, M. Alberti, M. Gavanelli, E. Lamma, P. Mello, and P. Torroni. The SOCS computational logic approach to the specification and verification of agent societies. 2004. This volume.
8. K. L. Clark. Negation as Failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.
9. Y. Dimopoulos and A. C. Kakas. Logic programming without negation as failure. In *Logic Programming, Proceedings of the 1995 International Symposium, Portland, Oregon*, pages 369–384, 1995.
10. P. M. Dung. On the acceptability of arguments and its fundamental role in non-monotonic reasoning, logic programming and n-person games. *Artificial Intelligence*, 77:321–357, 1995.
11. U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. The CIFF proof procedure for abductive logic programming with constraints. In *Proceedings JELIA04*. To appear.
12. U. Endriss, P. Mancarella, F. Sadri, G. Terreni, and F. Toni. Abductive logic programming with CIFF: implementation and applications. In *Proceedings CILC2004, Convegno Italiano di Logica Computazionale*, 2004.
13. T. H. Fung and R. A. Kowalski. The IFF proof procedure for abductive logic programming. *Journal of Logic Programming*, 33(2):151–165, Nov. 1997.
14. H. Haugeneder, D. Steiner, and F. McCabe. IMAGINE: A framework for building multi-agent systems. In S. M. Deen, editor, *Proceedings of the 1994 International Working Conference on Cooperating Knowledge Based Systems (CKBS-94)*, pages 31–64, DAKE Centre, University of Keele, UK, 1994.
15. Z. Huang, A. Eliens, , and P. de Bra. An Architecture for Web Agents. In *Proceedings of EUROMEDIA'01.* SCS, 2001.
16. J. Jaffar and M. Maher. Constraint logic programming: a survey. *Journal of Logic Programming*, 19-20:503–582, 1994.
17. A. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. A logic-based approach to model computees. Technical report, SOCS Consortium, 2003. Deliverable D4.
18. A. Kakas, P. Mancarella, F. Sadri, K. Stathis, and F. Toni. The KGP model of agency. In *Proceedings ECAI2004*, 2004. To appear.

19. A. C. Kakas, R. A. Kowalski, and F. Toni. The role of abduction in logic programming. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming*, volume 5, pages 235–324. Oxford University Press, 1998.
20. A. C. Kakas, P. Mancarella, and P. M. Dung. The acceptability semantics for logic programs. In *Proceedings of the Eleventh International Conference on Logic Programming, Santa Marherita Ligure, Italy*, pages 504–519, 1994.
21. A. C. Kakas and R. Miller. A simple declarative language for describing narratives with ations. *Logic Programming*, 31, 1997.
22. A. C. Kakas and P. Moraitis. Argumentation based decision making for autonomous agents. In J. S. Rosenschein, T. Sandholm, M. Wooldridge, and M. Yokoo, editors, *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS-2003)*, pages 883–890, Melbourne, Victoria, 2003. ACM Press.
23. A. C. Kakas and F. Toni. Computing argumentation in logic programming. *Journal of Logic and Computation*, 9:515–562, 1999.
24. A. C. Kakas, B. van Nuffelen, and M. Denecker. A-System: Problem solving through abduction. In B. Nebel, editor, *Proceedings of the 17th International Joint Conference on Artificial Intelligence*, pages 591–596, Seattle, Washington, USA, August 2001. Morgan Kaufmann Publishers.
25. R. Kowalski and F. Toni. Abstract argumentation. *Artificial Intelligence and Law Journal, Special Issue on Logical Models of Argumentation*, 4(3-4):275–296, 1996. Kluwer Academic Publishers.
26. R. A. Kowalski. *Logic for Problem Solving.* North-Holland, 1979.
27. R. A. Kowalski. Problems and promises of computational logic. In *Proceedings of the Symposium on Computational Logic*, pages 1–36. Springer-Verlag, 1990.
28. R. A. Kowalski and M. Sergot. A logic-based calculus of events. *New Generation Computing*, 4(1):67–95, 1986.
29. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd extended edition, 1987.
30. H. Prakken and G. Sartor. *A System for Defeasible Argumentation, with Defeasible Priorities*, pages 510–524. 1996.
31. H. Prakken and G. Sartor. Argument-based extended logic programming with defeasible priorities. *Journal of Applied Non-Classical Logics*, 7(1), 1997.
32. M. Shanahan. Prediction is deduction but explanation is abduction. In *Proceedings of the 11th International Joint Conference on Artificial Intelligence*, pages 1055–1060, 1989.
33. M. Shanahan. *Solving the Frame Problem.* MIT Press, 1997.
34. SICStus Prolog user manual, release 3.8.4, 2000. Swedish Institute of Computer Science.
35. K. Stathis, C. Child, W. Lu, and G. K. Lekeas. Agents and Environments. Technical report, SOCS Consortium, 2002. IST32530/CITY/005/DN/I/a1.
36. K. Stathis, A. Kakas, W. Lu, N. Demetriou, U. Endriss, and A. Bracciali. PROSOCS: a platform for programming software agents in computational logic. In J. Müller and P. Petta, editors, *Proceedings of From Agent Theory to Agent Implementation (AT2AI-4 – EMCSR'2004 Session M)*, pages 523–528, Vienna, Austria, 2004.
37. D. E. Steiner, H. Haugeneder, and D. Mahling. Collaboration of knowledge bases via knowledge based collaboration. In S. M. Deen, editor, *CKBS-90 — Proceedings of the International Working Conference on Cooperating Knowledge Based Systems*, pages 113–133. Springer Verlag, 1991.

38. F. Toni and K. Stathis. Access-as-you-need: a computational logic framework for flexible resource access in artificial societies. In *Proceedings of the Third International Workshop on Engineering Societies in the Agents World (ESAW'02)*, volume 2577 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 2002.
39. B. Traversat, M. Abdelaziz, D. Doolin, M. Duigou, J. C. Hugly, and E. Pouyoul. Project JXTA-C: Enabling a web of things. In *Proceedings of the 36th Hawaii International Conference on System Sciences (HICSS'03)*, pages 282–287. IEEE Press, 2003.

Author Index